Incremental view maintenance for Assemble by Anago

Maas, Rik

Award date:
2022

Department of Mathematics and Computer Science
Database Research Group

Incremental view maintenance for Assemble by Anago

Master Thesis

Rik Maas

Supervisors:
prof. dr. George Fletcher - TU/e
Fraser Wilson - Anago
ir. Thomas de Nooij MTD - Anago

Committee:
prof. dr. George Fletcher - TU/e
Fraser Wilson - Anago
ir. Thomas de Nooij MTD - Anago
dr. Renata Medeiros de Carvalho - TU/e

Final Version

Eindhoven, September 2022

Abstract

Incremental view maintenance (IVM) is the process of materializing query output and maintaining this output under updates on the input relations of the query, such that only the affected parts of the output are modified.

Assemble by Anago is software in which datasets are computed by the means of models, which are comparable to queries, as these also describe how an output relation can be created in terms of input relations. These models are based on *Oracle's OLAP DML*.

Assemble does not have mechanisms for efficiently updating output datasets when alterations are made to input datasets; if a small update has been made, the output sets can only be updated by running the entire model, thus for all output cells, even if these will not change. Therefore in this project we investigated whether *Dynamic Yannakakis* (DYN), a state-of-the art algorithm for dynamic query evaluation, is suitable for facilitating IVM for Assemble.

In order to examine this, we created the *Artificial Hour Administration* (AHA) application and *Availability calculations* (AC) model in Assemble. AC consists of a variety of types of sub-models, such that the most prominent types of sub-models in real models made by Anago are present. Also AC was made such that the number of sub-models is small, as to ensure that AC was feasible for this project.

The current version of DYN only facilitates query evaluation for a number of query operators, moreover no algorithms for computing delta relations were precisely specified. Therefore we extended DYN, such that each query corresponding to a sub-model in AC could be simulated and additionally procedures for computing the delta relations of these queries are described.

We show how the efficiency of the extended DYN algorithm compares to the efficiency of Assemble's model execution. From these results we have found that the extended DYN algorithm is significantly more efficient when indeed, as our use case defines, the update size is small and the total dataset size is big. This is because the number of iterated tuples in DYN depends on the update size, whereas for Assemble the number of iterated cells depends on the total dataset size.

However we also show that the current version of the extended DYN algorithm is not yet suitable for Assemble. Namely, for the foreseeable future, data will remain to be stored in an Oracle database, where its data representation is in terms of data cubes, i.e. multi-dimensional arrays. This type of data structure is not compatible with the current version of DYN, as it requires storing datasets in the format of a relation.

Acknowledgements

The past seven months I have been investigating incremental view maintenance, with special attention to the Dynamic Yannakakis algorithm, and the Assemble software. I want to thank some people who, during this period, have helped to bring this project to a successful end.

First I would like to thank my graduation supervisor George Fletcher, who introduced me to the Dynamic Yannakakis algorithm and throughout the project guided me with his feedback and advice.

Also I would like to thank the people at Anago who gave me the opportunity to do this research project and helped me getting familiar with Assemble. Special thanks goes to Fraser Wilson and Thomas de Nooij, my supervisors at Anago, for their feedback, advice and assistance in designing the AHA application and setting up experiments for Assemble.

Last but not least I would like to thank my friends and family for their support during the past five years, while I did my study program at the University of Technology in Eindhoven.

Contents

List of Figures

Chapter 1

Introduction

1.1 Background information

1.1.1 Anago

Anago in 's-Hertogenbosch is a company specialized in creating planning solutions for several organizations in the public, service and industrial sector [1]. However these solutions are not bound to planning only; for an organization they can provide any sort of overview or insight based on data provided by said organization, e.g. forecasts on sales or the company's turnover.

1.1.2 Assemble

The core of such solutions is *Assemble*. Assemble is a computer program that is developed, maintained and used by Anago. Using Assemble, employees of Anago are able to define *models* which compute the desired output datasets. A client of Anago is able to view these output datasets via an application which's content is fitted for their needs.

The storage of data, execution of computations and data manipulation in general is not done within Assemble itself. Rather this role is performed by the *Oracle OLAP Data manipulation language (DML)* on an *Oracle server*. The role of Assemble is that of providing an environment in which employees are able to define the datasets, in terms of properties such as the data type, attributes - in the Assemble terminology called *dimensions* - and models which are used to compute the contents of an output dataset, often in terms of input datasets. An employee can then choose to *distribute* the application, meaning that Assemble will then translate these definitions into commands for the OLAP DML, which the latter will execute and store the resulting datastructures in an *Oracle database*.

After the distribution has been performed, an application in the web browser will be able to retrieve the data from the Oracle database and therefore be visible for the client. However, the client is not only able to view the data. In contrast, depending on whether a dataset is read-only or not, one may be able to interact with the dataset, by changing dataset values. Often a client is able to let the Oracle server compute (updated) versions of the output datasets on demand, simply by clicking a button or closing a view.

1.2 Project goal

1.2.1 Problem statement

Incremental View Maintenance (IVM) is a technique in which query results, also called views, are materialized and maintained under database updates by computing and applying only the incremental changes i.e. computing and modifying only the affected parts of the view, instead of recomputing the query from scratch.

Assemble currently lacks such functionality, but could greatly benefit from it. Although the models in Assemble are not defined by the traditional query notation format, for example *SQL*, *datalog* or another language based on *relational algebra*, they can be seen as such, as both are able to access input data, manipulate it and create a new dataset. Moreover, the models in Assemble should be convertable to the *relational algebra* format.

Then the Assemble models also have resulting datasets which need to be maintained. A way in which Anago tries to combat this problem is by using *dynamic selections.* A dynamic selection is a selection on a dataset which can be changed for each time a sub-model result has to be computed. Often this selection is set only on updated cells, such that when executing a model it will only loop over dimension values that correspond to these updated cells.

However, this method may not work for certain models, since it cannot explicitly determine what output cells need to be updated. Also this method is far from optimal as in order to combat the previous downside, one is often forced to include extra dimension values in the selection in order to be certain that all ouput cells that need to be updated are updated. Due to these difficulties Anago is looking for an efficient IVM method, where this method would determine what cells are computed without the modeller having to be involved.

1.2.2 Dynamically update datasets in Assemble

In order to adapt Assemble such that datasets can be efficiently updated dynamically, one can look at the current stage-of-the-art research performed on IVM and search for a result that seems promising and fitting to Assemble models.

For this project one such result it chosen. Then the goal of this project will be to find out whether this result is suitable for Assemble models. This is done by creating an adapted implementation for this result and a specific Assemble model, where the performance of this implementation can be compared to that of the original Assemble model.

This chosen result is the *Dynamic Yannakakis* (*DYN*) algorithm by Idris et al. [8]. The idea behind this algorithm is to derive from the query a *General Join Tree* (GJT), which describes a high-level query plan, where it states the query operations and the (partial) order in which these take place. Then the algorithm will convert this GJT to a *Tree Reduction* (T-reduct). This entails that an initial *update* procedure is executed, which will first extend the leaves of the GJT with a relation ρ, each being one of the input relations. Then, in a bottom-up manner, it will also generate relation ρ at the inner nodes of the tree. After the update procedure has finished, the tree and the relations in the nodes, by an *enumeration* procedure is able to produce the output relation of the original query in an efficient manner.

Afterwards, if there are updates in the input datasets, one can again execute the update procedure, such that the T-reduct, specifically ρ for each node, is updated in an efficient manner. Also, the enumeration procedure will now produce the updated output relation.

The *DYN* algorithm in particular is chosen for this project because:

1. It is competitive with state-of-the-art results on dynamic query evaluation in terms of memory and time complexity
2. Practicality: on a conceptual level it is rather straightforward, which makes it relatively easy to implement

1.2.3 Research questions

The main research question that this project will try to answer is:
What does the Dynamic Yannakakis algorithm offer in terms of realizing IVM for Assemble mod-

els?

This will be broken down in subquestions:

1. What does Dynamic Yannakakis in it's current state offer?
2. Can we extend Dynamic Yannakakis to facilitate IVM for Assemble models, without nullifying the benefits of the current Dynamic Yannakakis algorithm?
3. Considering the way Assemble is intertwined with the Oracle OLAP DML, would Dynamic Yannakakis be a practical short-term solution?

For this project we choose to focus on one Assemble application and model, which has been tailor made for this project. Hence it is far from guaranteed that this model and application covers all situations. This means that there may be model functionality that is not considered in this project. This has to be taken into account for the stated questions, in order for the reader to realize, that if this project is successful in showing that Dynamic Yannakakis works for this model, a full working implementation that accomplishes IVM for any Assemble model may still be far away. However the model used in this project is made with the purpose of simulating the most prevalent types of functionality in general Assemble models. Therefore it should still give a good idea about the potential of DYN.

Despite this drawbacks of focussing on one application and model, it does offer the following benefits:

- We can let the sub-models functionality reflect the most prevalent functionality in real Assemble models
- Certain parts of this project need to be carried out manually, thus limiting the scale improves feasibility
- The results of the final implementation for this project can be compared more easily to the Assemble application

1.2.4 Measuring performance

In order to determine the performance of the algorithm, one should therefore find a performance metric that is fair, in the sense that it is not dependent on factors that lie outside the algorithmic performance. Therefore taking as a metric the *time* that the implementation takes for processing an update, is out of the question.

A fair metric is to determine the number of cells - or an equivalent concept in the implementation - that needed to be accessed for a certain update, model, input dataset(s) and output dataset. One can verify that this is a fair metric, since the number of accessed cells could be derived from the algorithms without needing the implementation of these algorithms. However since verifying the correctness of the algorithms and determining the number of cells is the most feasible by making an implementation, this is still the approach that has been followed.

1.2.5 Thesis outline

In the remainder of this chapter a literature analysis will be done. Here we look at the progression of DYN and relevant work on IVM and dynamic query evalution in general.

Chapter 2: Preliminaries, gives a more in-depth description of the concepts in Assemble, with the goal that an intuition is given for what the types of submodel functionality are, which need to be taken into consideration later when DYN is discussed.

In chapter 3: Models and methods, first a description is given of the *Artificial Hour Administration* application and *availability calculations* model, secondly we analyze the behaviour of an update, and thirdly we look at how AC can be described in terms of relational algebra, in order to be suitable for DYN.

Then, in chapter 4: The current state of Dynamic Yannakakis, we take a look at the current state of DYN. Here we identified what the current version of DYN facilitaties in order to realize IVM for the AHA application, but also what is still needed for this purpose.

This chapter is followed by chapter 5: Extending Dynamic Yannakakis, in which algorithms and procedures are given to extend DYN, such that it facilitates the missing functionality. Additionally we show that this extended version is still not suitable for the current needs of Anago, as the data representation of stored data in an Oracle database does not correspond to the way in which DYN accesses data that needs to be stored.

Chapter 6: Experiments, compares the efficiency of DYN to that of AC executed in Assemble. Also it shows that indeed DYN's delta enumeration procedure's efficiency does not depend on the total relation size.

In chapter 7: Conclusion, we answer the research question and sub-questions, give a recommandation for Anago on how they can follow up on the work done in this project, and look at contributions made by this project to research on dynamic query evalution.

Finally, in chapter 8: Future work, we discuss how the research done during this project can be continued for both DYN in general and DYN, but also IVM in general, for Assemble.

1.3 Literature analysis

1.3.1 Progress on Dynamic Yannakakis

DYN is based on Yannakakis' algorithm for computing results of acyclic join queries [13]. Unlike DYN, the algorithm by Yannakakis is aimed at the static setting.

Idris et al. modified Yannakakis' algorithm to work in the dynamic setting, thus for computing acyclic join queries under updates, and published the resulting algorithm Dynamic Yannakakis (in short D_{YN}, where we refer with D_{YN} to this specific version of the algorithm, unlike DYN) in 2017 [5].

Since, Idris et al. have been expending upon D_{YN}, such that these can be used in more general cases. Namely, D_{YN} only worked for acyclic join-queries which contain only equivalences in their join predicates.

Therefore, in 2018 Idris et al. updated D_{YN}, where the results are refered to as GD_{YN} and IE_{YN} [6]. GD_{YN} can process updates for queries with any join predicate in log-linear time and enumerate query results with logarithmic delay and IE_{YN} improves on GD_{YN}, such that query results can be enumerated in constant delay iff each join predicate contains at most one inequality.

Where [6] only sketched why GD_{YN} and IE_{YN} work correctly, in [7] Idris et al. give a formal proof for the correctness of the algorithms. Additionally a novel algorithm for computing GJT's is given and it's correctness is illustrated.

1.3.2 Related work

Incremental view maintenance

IVM in general has been a widely researched topic in the database domain.

Chirkova et al. conducted a survey on the topic of materialized views [4]. Here materialized views refer to the results of queries that are materialized and maintained to facilitate access to base tables. Then the survey focuses on how (e.g. in what data structure) to maintain and use such views, and how to decide what intermediate results to maintain.

Lee et al. have proposed a method for efficient maintenance of data cubes [11]. More specifically, for a relation R($\overline{a}$) with dimension set $\overline{a}$, then there exists a number of relations that are aggregations over a subset of dimensions $\overline{a}$, which we call views. Then there are $2^{|\overline{a}|}$ such views.

Then this method increases efficiency if several of these views need to be computed for such a relation R. Namely traditional methods require for each view to compute the delta relation, whereas the method by Lee et al. is a heuristic method where only $\binom{n}{\lceil|\overline{a}|/2\rceil}$ delta relations need to be computed.

Dynamic query evaluation

Also dynamic query evaluation in general has been a widely researched topic in the database domain. Just as with DYN it is not always the case that a query result is materialized and incrementally maintained, rather a dynamic datastructure is proposed which allows insertions, deletions and efiicient enumeration of the output of the query.

Where DYN considers conjunctive, acyclic queries, research also has been done on other type of queries. Now we will discuss such research results.

Kara et al. investigated the trade-offs in static and dynamic evalution for hierchical queries [10]. Consider query Q and the set of atoms in Q, atom(Q). Then for a variable $x \in$ var(Q), atom(x) is the set of each $A \in$ atom(Q), such that $x \in A$. Then Q is hierchical if for any two variables $x, y \in$ var(Q) we have atom(x) $\subseteq$ atom(y) $\vee$ atom(x) $\supseteq$ atom(y) $\vee$ atom(x) $\cap$ atom(y) $= \emptyset$.

For dynamic evaluation the trade-off made was between *preprocessing time*, which is time needed to compute the data structure that represent the query result, *update time*, which is the time to update the data structure under updates to the input data, and finally *enumeration delay*, which is the time between the retrieval of two tuples in the output relation.

Kara et al. have investigated the problem of incrementally maintaining triangle count queries under updates [9].

Such a query is opposed to queries considered by DYN cyclic: namely it considers queries in the form $R(a, b) \bowtie S(b, c) \bowtie T(c, a)$. More specifically a count query computes:

$\sum_{a \in Dom(A)} \sum_{b \in Dom(B)} \sum_{c \in Dom(c)} R(a, b) * S(b, c) * T(c, a)$,

where $dom(X)$ is the set of dimension values for dimension X.

Berkholz et al. investigated dynamic query evaluation for unions of conjunctive queries ($UCQs$) [3]. They proposed a data structure which can not only enumerate the query output, but can also check whether it contains a certain tuple or count the size of it.

Chapter 2

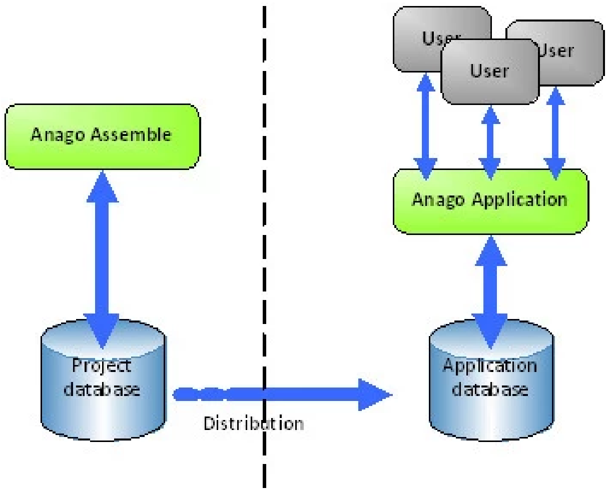
Preliminaries

2.1 A closer look at Assemble

2.1.1 Anago application architecture

The Anago application architecture is shown in figure 2.1 and is accompanied by the following description. An Anago application can be devided into two parts, namely the Anago Assemble environment and the Anago (User) application environment. The Assemble environment is used by modellers as a place to build a so called project. This project is stored into the project database and can be distributed, such that in the user application environment the application database is updated, which stores the user application data. Then users can interact with the application by entering, calculating and viewing data, via the internet browser. From now on we will refer to the user application (environment) with *(user) application* and to the Assemble project (and environement) with *Assemble*.

Figure 2.1: Anago application architecture



Views

A user application consists of one or more *views*. A view is a page in which a user is able to interact with data and contains mainly tables and diagrams, where the former may be editable and the latter is always read-only. The data displayed in these tables are stored in *datasets* as descibed in section 2.1.2.

Non-editable data may be inserted in the Assemble environment, either manually by the modeller or via a *connection*, which is a way for the modeller to extract data from another source (for example a text or excel file), and load them into an Assemble dataset.

2.1.2 Data model

Dimensions

In Assemble data attributes are called *dimensions*. Each dimension defines a range of values, called *dimension values*, which are used for indexing data entries, where the latter are called *cells*.

A dimension can be flat or hierarchical, where the latter means that a dimension consists of 2 or more *levels*. Here a dimension value on a level is always mapped to a dimension value on the level above (except for dimension values on the highest level).

A special case of a dimension is the *time* dimension. A time dimension contains time values such as *day*, *week* and *year*. A start and end date can be defined, as well as the time levels. These levels are days, years and either months, quarters and halfyears or weeks and periods.

Something to point out is that inside Assemble (or the application), levels inside a hierarchical dimension are not distinguished by a per level name. In contrary, an hierarchical dimension often has a name that seems to specify only the 'most dominant' level ('most dominant' being a bit arbitrary, but often there is a level which is used the most). However, it does then still apply to any value in the dimension. As example, a dimension can be called $week$, which often has next to level $week$ also level $year$. Yet, to all dimension values in this dimension, hence also to those of level year (and period), is refered by dimension name week.

Datacubes and relations

Assemble uses two different kinds of database objects to store data. These are *datacubes* and (many to one) *relations*.

A datacube is defined over zero or more dimensions, where a cell exists for each combination of values over these dimensions. Then also a datacube's datatype should be specified, which can be either an *integer, decimal, boolean* or *text*, as well as - although rarely used - *short integer* or *short decimal*.

A relation defines a connection between one or more source dimensions and a target dimension.

From now on, let a datacube or relation with name D and dimension set or source dimension set respectively $\overline{x}$ be refered to as $D(\overline{x})$. (Or simply D when the dimensions are not relevant.)

Both datacubes and relations have an (boolean) option *autofill datacube/relation based on time*. Enabling this has as result that if the user in the application (manually) enters a value x in a cell of a datacube or relation - for which one of its dimensions is a time dimension - then, assuming that this cells time dimension value is t and the other dimension values for this cell is defined by set A, all cells with dimension values A and time dimension values $t+1$...the end date, get value x as well.

Datasets and selections

Whenever a datacube or relation is to be used, which may either be in a model or a view, a *dataset* should be defined. In this dataset a datacube or relation is specified, as well as a *selection* over each dimension of a datacube or each source dimension in a relation. For simplicity reasons, we refer to a dataset with the datacube or relation it is defined by, and additionally by the selection where this is relevant. For example, we refer to a cell of dataset $D(a, b, c)$ as *D(a 'x', b 'y', c 'z')*, where x, y and z are dimension values for dimensions a, b and c respectively.

For a dimension d, we say that a selection rule S defines a set of dimension values V' over $var(d)$ - where $var(d)$ is the total set of dimension values of d and $V' \subseteq var(d)$ - by writing $S(var(d)) \rightarrow V'$. A selection rule S is composed of a sequence S_1 $(method\ S_i)^k$ such that $S_1(V)$ $(\diamond\ S_i(V))^k \rightarrow S(V)$, $k \geq 0$, $method \in$ {add, keep, remove} corresponding to set operators $\diamond \in \{\cup, \cap, \setminus\}$ respectively. Hence a selection rule consisting of at least rule S_1 followed by zero or more method-rule pairs. Then each rule (including R_1) can either be another sequence of rules and methods, or a selection rule defined by predicate p.

p can be specified in several manners:

- When the dimension values are defined inside Assemble itself, one or more of these values can be selected.
- When a dimension is hierarchical, to select one or multiple levels and consequently selecting the values belonging to this/these level(s).
- A selection can be defined over a relation R, when (one of) the source dimension(s) of R is d. Then one or multiple values x, $x \in var(o)$, o being the target dimension of R, can be chosen, such that values $v \in var(d)$ are selected for which vRx holds.
- A so called *dynamic* selection S can be defined, refering to the fact that S is based on a current time period (for example current week or year) or on cells corresponding to user changed values in the application, such that $S(var(d))$ changes.
- A *relative* selection rule s can be only applied to hierarchical levels and can be defined in terms of another selection rule s'. Then s can be based on the relative hierarchy or position to s'. For example one can define s' as selecting the current week, then one can define s to select the year corresponding to the current week (hence current year), and the 2 years preceding and 3 years succeeding this current year.
- An *expression* selection can be applied on d in the following manner: choose a dataset $D(d, x')$, x' being either an empty set or a set with dimensions for which each selection chosen for D selects only one value. After such dataset is chosen, each dimension value in d should map to one value in D. Then an operator $\circ \in \{=, <, \leq, >, \geq\}$ and value v can be chosen such that dimension value set $V' \subseteq var(d)$ is selected where $i \in V' \iff D[i] \circ v$.

Models

A *model* can be seen as a relation between a number of input datasets and output datasets. It is defined by a sequence of calculations, each of which refered to as *submodel*. Each submodel has 0 or more input datasets and based on those determines the values for the output dataset.

A model's execution is initiated from the application environment; either when opening, saving or closing a view, by clicking a button or after changing a cell's value. Then submodels are executed in the exact order of the sequence. This is important because results of a submodel may be used by a submodel later in the sequence. Hence for a model M we define a total order on the submodels in M, such that $\forall_{x,y \in M, x \neq y}(x < y \vee y < x)$, $x < y$ meaning that submodel x is executed before submodel y.

Each submodel is either a standard calculation or a free calculation.

A standard calculation is a predefined computation with a fixed number of input datasets. The following standard calculations are most prevalent:

1. Aggregator - Used to aggregate the values of one dataset to another dataset based on 2 input datasets, a datacube and a 1-to-1 relation. E.g. for data cube *D(a, b, c)* and relation *R(a, d)* the aggregator calculation will compute a datacube *D'(d, b, c)*.
2. Roll-up - Used for a data cube for which one dimension is hierarchical, where the cube contains the data on the lower level but not on the level above this level. Then the values of the data cube corresponding to the lower level will be rolled-up to the higher level. E.g. for a data cube *D(a, b, time)*, where *time* consists of the levels *month* and *year*. Then if the cube contains information on every value of a, b and *month*. The model will compute the value for each a, b and *year* by summing over all *months* corresponding to each *year*.

3. Common statistics - Used for calculating common statistical functions. The complete list of such functions is: *sum, average, standard deviation, variance, minimum, maximum count* and *average - (minimum + maximum)*. This sub-model has one input data cube and one output data cube. The dimensions of the output data cube must be the same as the dimensions of the input data cube where at least one dimension has to be left out. Then the function will compute the statistical function over the dimension(s) that is/are left out.

A free model is defined over an output dataset, 0 or more input datasets and an expression. In order to control which cells in the output dataset are to be computed, a selection on the output dataset can be set, but this must be done in the dataset definition. The selections on the input datasets must be equal to those of the output dataset for corresponding dimensions, even if values outside the dimension selection are used as input, which is possible because selections on input datasets do not determine what cells can be used - or are used - for input. The expression is an instruction based on the Oracle OLAP DML on how the output should be computed. Although it must adhere to a predefined syntax, the expressions are very flexible. A free model expression consists of the following components:

- References to the input dataset(s).
- Mathematical operators/precedence symbols - The complete list is: + (addition), - (subtraction), * (multiplication), / (division), ** (power) and (...) (priority brackets). As example for data cubes *D1(a, b)* and *D2(a, b)*, D1 + D2 will compute data cube *D3(a, b)*, where each cell *D3(a 'x', b 'y')* has value *D1(a 'x', b 'y') + D2(a 'x', b 'y')*.
- Logical operators - The compute list is: eq (equals), ne (not equals), gt (greater than), ge (greater or equals), lt (less than), le (less or equals), not, and, or. Can be used in combination with at least one data cube and returns a boolean. Therefore it is often used in combination with conditional expressions.
- Conditional expression - Can be created by an if-then-else construction. E.g. consider input datasets *D1(a, b, c), D2(a, b, c), D3(a, b, c)* and output dataset *D4(a, b, c)* and expression:
  if D1 gt 3 then D2 else D3,
  will fill *D*4 where a cell *D4(a 'x', b 'y', c 'z')* will get the value of *D2(a 'x', b 'y', c 'z')* if *D1(a 'x', b 'y', c 'z') > 2* and else *D3(a 'x', b 'y', c 'z')*.
- Oracle OLAP DML functions. Some examples are max(Expression1, Expression2), min(Expression1, Expression2) and nafill(x, y). Here Expression1 and Expression2 can be either datasets or numbers. nafill(x, y) will go over dataset x and each value that is na (not available) will be replaced by y. A very common usage in Assemble of nafill is with y = 0. The complete list of functions can be found at the OLAP DML reference [2].

Chapter 3

Models and methods

3.1 Artificial Hour Administration application

The *Artificial Hour Administration* (AHA) application is an Assemble project and application, specially made for this research project, which mimics the behaviour of an hour administration application.

In this section a general description of the AHA application will be given.

3.1.1 Views

The AHA application consists of 2 views.

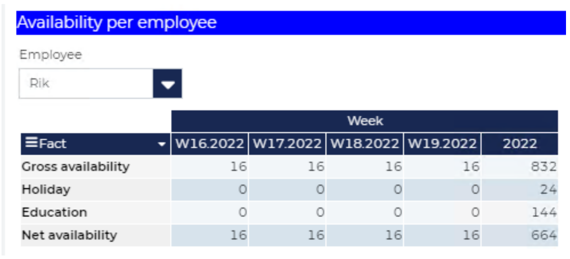
Availability view

The *Availability* view consists of 2 parts. The first part contains a table in which an employee can write education and holiday hours on a weekly basis, where the current and following 3 weeks are displayed. It is shown in figure 3.1. The dimension on the vertical axis is called the *Fact* dimension, having dimension values *gross availability, education, holiday* and *net availability.*

Gross availability describes how many hours an employee works without considering time necessary for holiday and education. Holiday and education describe the amount of hours an employee uses for holiday and how many hours are spent on education purposes respectively. These are meant to be updated in this table by the respective employee. Net availability describes how many hours an employee works when the education and holiday hours are subtracted from the gross availability. Net availability will be computed by a model.

Finally for each fact and employee the number of hours is also shown for the current year, which is also computed by a model.

Figure 3.1: Hours per employee per fact per week



Availability per employee

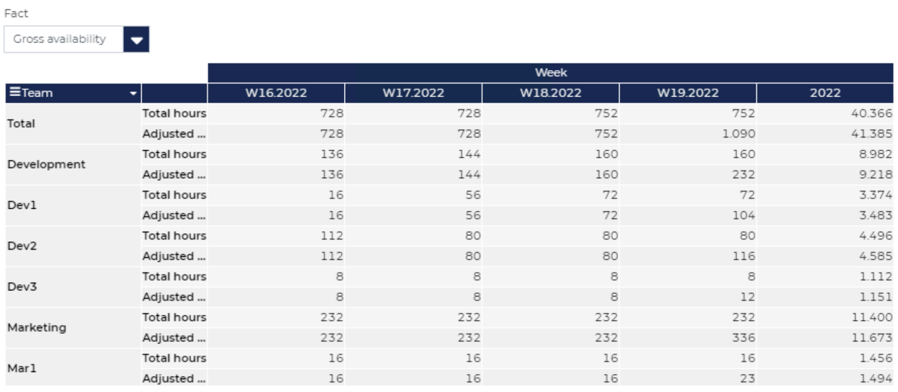
Employee: Rik

	Week				
Fact	W16.2022	W17.2022	W18.2022	W19.2022	2022
Gross availability	16	16	16	16	832
Holiday	0	0	0	0	24
Education	0	0	0	0	144
Net availability	16	16	16	16	664

The second part of the view contains tables which shows data on a per team basis. It consists of 3 tabs.

The first tab shows the total number of hours, as well as *adjusted* hours (which are computed based on an percentage), per fact, team and week (again the current week plus the 3 following weeks) and for the current year. This view is partly shown in figure 3.2. Team is a hierarchical dimension, consisting of levels *team, department* and *total.* The hours and adjusted hours are shown not only on the team level, but also on the department and total level. These are such that each team belongs to a department, and all departments together make up total.

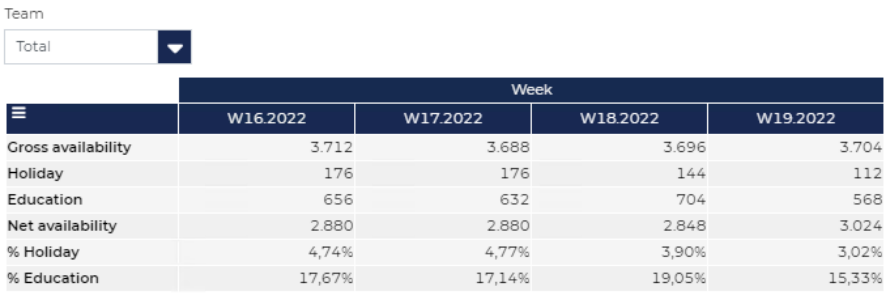
Figure 3.2: Hours and adjusted hours per team per fact per week


Fact: Gross availability

☰Team		Week				
		W16.2022	W17.2022	W18.2022	W19.2022	2022
Total	Total hours	728	728	752	752	40.366
	Adjusted ...	728	728	752	1.090	41.385
Development	Total hours	136	144	160	160	8.982
	Adjusted ...	136	144	160	232	9.218
Dev1	Total hours	16	56	72	72	3.374
	Adjusted ...	16	56	72	104	3.483
Dev2	Total hours	112	80	80	80	4.496
	Adjusted ...	112	80	80	116	4.585
Dev3	Total hours	8	8	8	8	1.112
	Adjusted ...	8	8	8	12	1.151
Marketing	Total hours	232	232	232	232	11.400
	Adjusted ...	232	232	232	336	11.673
Mar1	Total hours	16	16	16	16	1.456
	Adjusted ...	16	16	16	23	1.494

The second tab shows a *moving total* of the hours per team, fact and week, as displayed by figure 3.3. More specifically; for each team and fact, it shows for a week the sum of the (non-adjusted) hours, together with the hours of the 4 preceding weeks. Next to this 2 extra facts are added; *% Holiday* and *% Education*, which show the relative moving total of holiday and education respectively, to the gross availability.

Figure 3.3: Moving total per team per fact per week


Team: Total

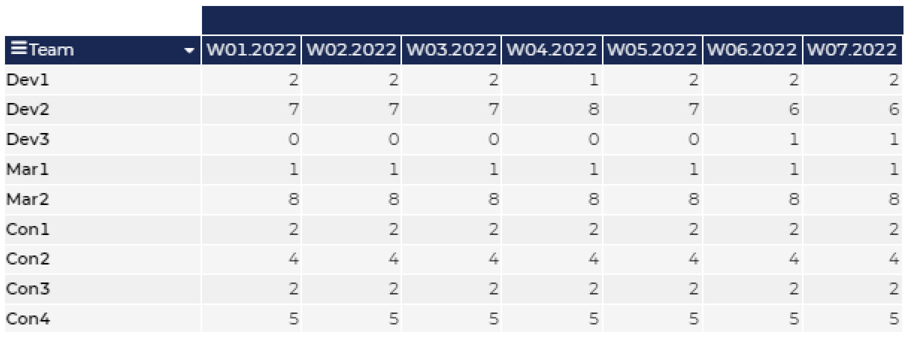
☰	Week			
	W16.2022	W17.2022	W18.2022	W19.2022
Gross availability	3.712	3.688	3.696	3.704
Holiday	176	176	144	112
Education	656	632	704	568
Net availability	2.880	2.880	2.848	3.024
% Holiday	4,74%	4,77%	3,90%	3,02%
% Education	17,67%	17,14%	19,05%	15,33%

The third tab shows the number of employees per team and week, where only the teams on the team level are shown. A screenshot is shown in figure 3.4.

Configuration view

The *Configuration* view has as its main purpose to configure information. Figure 3.5 contains a screenshot of this view. In terms of configuration, the view contains a table for selecting a team for each employee and week, the gross amount of hours that an employee will work each week and a so called 'adjustment percentage' that is defined for each fact and week. Finally there is a table showing whether a team is empty for each week. This is the only table in this view which the user cannot alter. Rather the model, described in the next section, determines the content of this table.

Figure 3.4: Number of employees per team and week



≡Team	W01.2022	W02.2022	W03.2022	W04.2022	W05.2022	W06.2022	W07.2022
Dev1	2	2	2	1	2	2	2
Dev2	7	7	7	8	7	6	6
Dev3	0	0	0	0	0	1	1
Mar1	1	1	1	1	1	1	1
Mar2	8	8	8	8	8	8	8
Con1	2	2	2	2	2	2	2
Con2	4	4	4	4	4	4	4
Con3	2	2	2	2	2	2	2
Con4	5	5	5	5	5	5	5

3.1.2 Availability calculations

The Assemble project contains one model called *Availability calculations* (AC), containing 14 submodels. Figure 3.6 shows the model by means of a *directed acyclic graph (DAG)*.

Each node in the graph either respresents a submodel - more specifically its output dataset - or an input dataset to a submodel (if it is not an output dataset of another submodel). From now on note that I will refer to a node as a submodel or dataset when applicable. The datasets that coincide with blue nodes are datasets to which a user can make changes. Datasets that coincide with red nodes are ones that the user will see in the view, however this also applies to the blue nodes, since if a user makes an update to a table this is done inside a view. A distinction that red nodes do have is that those are output datasets of AC.

Each node contains 3 parts, an upper part used for identification and the type, a middle part that shows the dimensions, and selections on the output dataset, or just the dataset if the node is an input dataset. For submodel nodes the bottom part shows the computation; either a free model's expression or a description of a standard model's operation.

The upper part of a node is constructed in the following way. The dataset for each node is either a relation (not to be confused with a relation as we know it in relational algebra) or a datacube. Datacube nodes start with an expression $<< DataType >>$ where $DataType$ is the datatype, hence *integer, decimal, boolean* or *text.* For relations this datatype can best be compared to an enumerator over the values of the target dimension, but is not further specified. Then for each node an identifier is assigned. This is either a number 1-14, if the node coincides with a submodel or an expression ik, $k \in \mathbb{Z}$, if the node coincides with an input dataset only. Finally the datacube or relation is described by stating DC or R respectively, followed by its Assemble identification number. Then for datacubes a descriptive name of its cells' values is given.

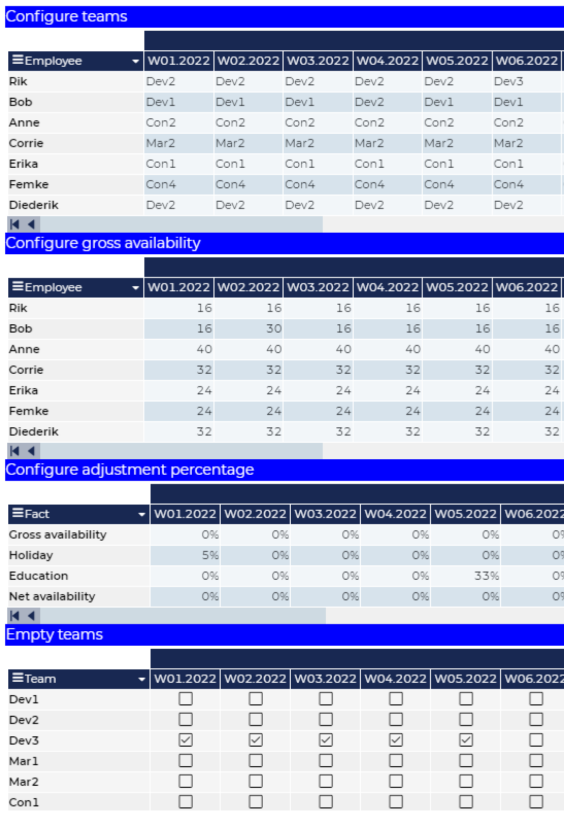
An edge $(d1, d2)$ indicates that:

1. The datacube/relation defining dataset $d1$ is used as input to submodel $d2$
2. $d1 < d2$, thus submodel $d1$ is executed before submodel $d2$
3. It is *likely, but by no means certain* that values in dataset $d1$ are used by submodel $d2$

A dashed arrow is used when the source and target of the edge contains the same datacube/relation. A solid arrow is used otherwise.

Note that an edge $(d1, d2)$ for submodels $d1$ and $d2$, does *not* define that output dataset $d1$ is used as input dataset by model $d2$, where $d1$ is the *specific* output dataset of submodel $d1$. A first reason why this is not possible is that often there is a mismatch between dataset selections. Consider edge $(m1, m2)$. Say we there are output datasets $D1(\underline{a}, x1)$ and $D2(\underline{b}, x2)$ for submodels $m1$ and $m2$ respectively. Here, a and b are dimension sets with $a \cap b \neq \emptyset$ and selection predicates

Figure 3.5: Configuration view



Configure teams

Employee	W01.2022	W02.2022	W03.2022	W04.2022	W05.2022	W06.2022
Rik	Dev2	Dev2	Dev2	Dev2	Dev2	Dev3
Bob	Dev1	Dev1	Dev1	Dev2	Dev1	Dev1
Anne	Con2	Con2	Con2	Con2	Con2	Con2
Corrie	Mar2	Mar2	Mar2	Mar2	Mar2	Mar2
Erika	Con1	Con1	Con1	Con1	Con1	Con1
Femke	Con4	Con4	Con4	Con4	Con4	Con4
Diederik	Dev2	Dev2	Dev2	Dev2	Dev2	Dev2

Configure gross availability

Employee	W01.2022	W02.2022	W03.2022	W04.2022	W05.2022	W06.2022
Rik	16	16	16	16	16	16
Bob	16	30	16	16	16	16
Anne	40	40	40	40	40	40
Corrie	32	32	32	32	32	32
Erika	24	24	24	24	24	24
Femke	24	24	24	24	24	24
Diederik	32	32	32	32	32	32

Configure adjustment percentage

Fact	W01.2022	W02.2022	W03.2022	W04.2022	W05.2022	W06.202
Gross availability	0%	0%	0%	0%	0%	0
Holiday	5%	0%	0%	0%	0%	0
Education	0%	0%	0%	0%	33%	0
Net availability	0%	0%	0%	0%	0%	0

Empty teams

Team	W01.2022	W02.2022	W03.2022	W04.2022	W05.2022	W06.202
Dev1	☐	☐	☐	☐	☐	☐
Dev2	☐	☐	☐	☐	☐	☐
Dev3	☑	☑	☑	☑	☑	☐
Mar1	☐	☐	☐	☐	☐	☐
Mar2	☐	☐	☐	☐	☐	☐
Con1	☐	☐	☐	☐	☐	☐

$p1$ and $p2$ respectively, where $p1 \neq p2$ for at least one dimension in $a \cap b$. Then since $(m1, m2)$ is an edge, $D1$ would be an input dataset for $m2$, but this is not possible since there should not be a mismatch between the selections of input and output datasets in a submodel. Hence $D1$ cannot be used as dataset in $m2$.

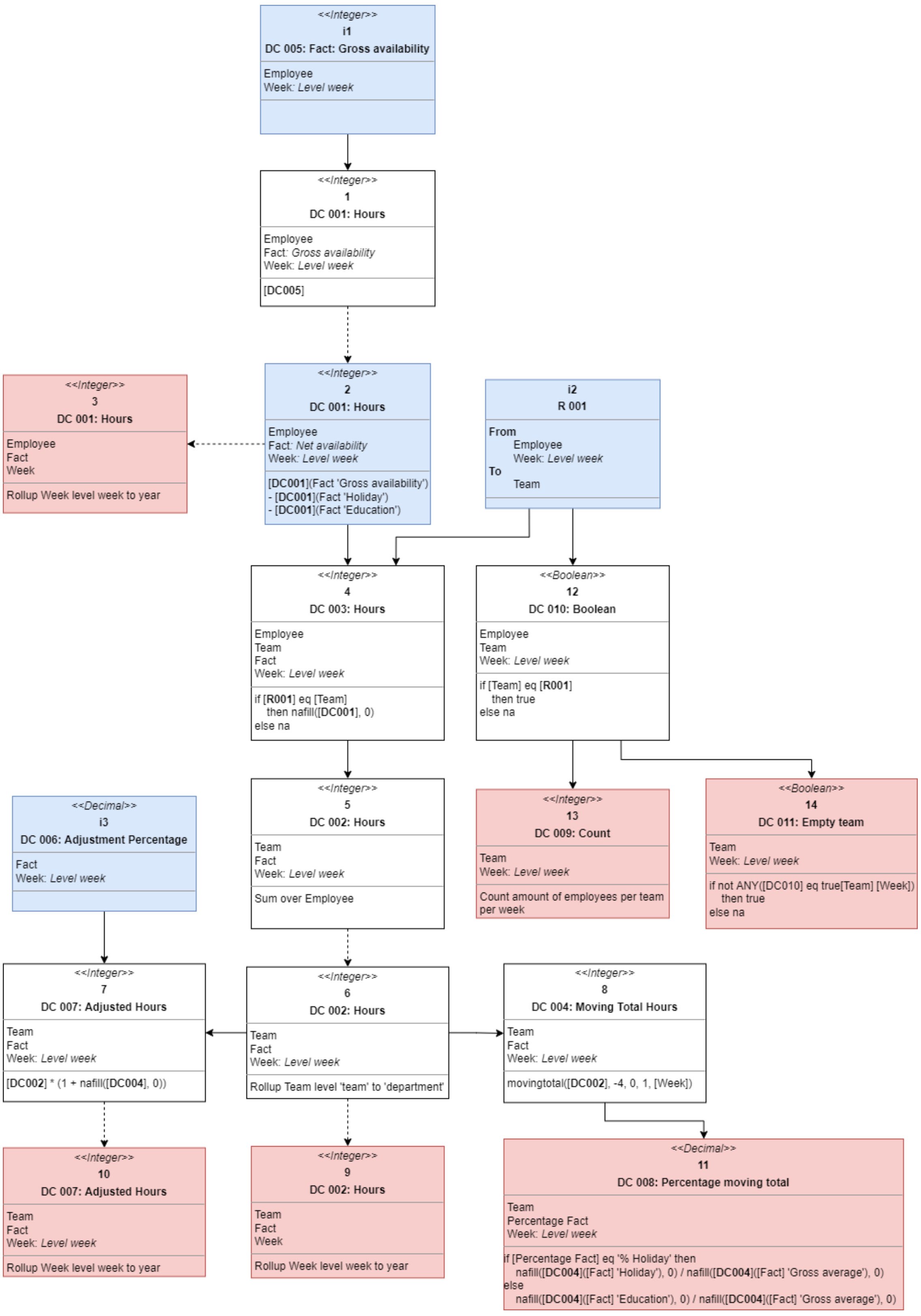
Another reason why this extension of the definition does not work is that for a submodel m with Assemble output dataset d, d can be used as input dataset for m, which would result in an edge (d, d). Now this is not possible because of definition 2.

AC is designed in such a way that, although 14 submodels is quite a low number, the most prevalent submodel functionality is present in the model. Concretely, the most prevalent submodel functionality includes, but is not limited to, together with the node names in which the functionality is used in parentheses:

- The datacube/relation or dataset perspective:
  - Autofill on the datacube/relation (i2, i3)
  - Selection on the output dataset (almost all, but specifically 2)
  - Datacube datatypes; integer, decimal, boolean and text (all datacube nodes)
- Standard calculations:
  - The roll-up operation (3, 6, 9, 10)
  - The common statistics, in this case with the sum and count function (5, 13)
  - The aggregator operation, albeit indirect (4 + 5)
- Free calculations:

  - Mathematical operators used to combine values from different datasets (2, 7, 11)
  - Conditional expressions, containing logical operators (4, 11, 12, 14)
  - Oracle OLAP DML functions, present functions are *nafill, movingtotal* and *any* (4, 7, 8, 11, 14)

Figure 3.6: Availability calculations model



3.2 Update analysis

3.2.1 General description of analysis

In this section an analysis is performed on the propagation of two updates through AC, where the behaviour is descibed per update for each sub-model. For each update step two properties are important:

1. When the step is applied, the output dataset is exactly as it would be when the entire model would be executed.

2. Executing the step is efficient, meaning that the step has a minimal, or at least low number of input and updated cells.

For each sub-model a description of the update step is given. The description entails a list of used variables and how they are used in the update steps. Then an iteration over several ranges of variables is given, denoting what are the updated cells. Thirdly the update methods are given in the form <*updated cell, update value*>. The first of the update methods always uses the same input datasets as the actual sub-model does, where a selection on the dimension values is put, such that only the necessary input cells are consulted. The second method does not use the same input datasets, but rather uses the previous value of the output dataset, the initial update value (x) and other necessary cells.

The analysis entails statistics on the number of cells that are used as input, and the number of cells that are updated. For this model, for each sub-model the number of updated cells per method is the same, namely only the cells that do change. Then the number of input cells is always the number of update steps, multiplied by the variables in each such step. Note that for both updated and input cells, these are the minimal number needed where the actual number depends on the implementation of an update method.

3.2.2 Purpose

The reason this update analysis is performed is not very specific. The initial idea arose from the motivation for this project; namely propagating updates would be more trivial when deciding input and output cells for a submodel could be done in an automatic manner. Because then the update process would entail determining these cells and then apply the calculations on, and using these cells.

Now determining these cells automatically is not trivial. Perhaps it is managable, but still then it will not be the main strategy for how this project will attempt to apply updates, since the algorithms that are investigated do not do this either.

Although not automatical, determining the input and output cells of submodels is managable manually, at least when the understanding of the specific submodels and dataset (selections) is large enough. This is the case for the AHA model. Then determining these cells and deriving update steps may lead to an alternative update strategy, which the resulting algorithm(s) of this project can be compared to, and be used for testing these. Therefore, next to enhance the understanding of the model, this analysis does have a concrete goal in being a first step in the process of creating test algorithms.

3.2.3 Update descriptions

For the first update we consider arbitrary employee a and week $w_{b.y}$. Here b is the week number and y the year number, where we make the assumption that the data set contains data for year y and $y+1$, containing $len(y)$ and $len(y+1)$ weeks respectively. For the second update this is the same, except for the fact that weeks are refered to by a single integer. This is merely for simplicity reasons, the actual model still holds weeks in format either $w_{i.j}$ for cells refering to weeks on level week and k for weeks on level year. The following definitions are assumed in this analysis:

- Let t be the team corresponding to employee a and week $w_{b.y}$, as described by relation $R001$
- Let team D be the department to which team t belongs
- Let team T be the total of all departments
- For a hierarchical dimension value x, that is on level l, $l > 0$, let $set(x)$ denote the set containing cells on level $l-1$ that correspond to x. For example $t \in set(D)$ holds.

For both updates an example is shown. In figures 3.7 and 3.8, the number of updated cells and input cell per method are shown for both updates. The remaining update descriptions can be found in the appendix.

Update 1

Update: *<DC_001(Employee 'a', Fact 'holiday', Week '$w_{b.y}$'), x>*
This update is applied to datacube *001 Hours per fact per employee per week* and changes the cell at Employee a, Fact $Holiday$ and Week b to integer x.

For the remainder of this update:

- Let x' be the previous value of *DC_001(Employee 'a', Fact 'holiday', Week '$w_{b.y}$')*

2 - Compute net availability per employee per week
Definitions

- Let c be *DC_001(Employee 'a', Fact 'gross availability', Week '$w_{b.y}$')*
- Let d be *DC_001(Employee 'a', Fact 'education', Week '$w_{b.y}$')*
- Let e be the previous value of *DC_001(Employee 'a', Fact 'net availability', Week '$w_{b.y}$')*

Update methods

1. · *<DC_001(Employee 'a', Fact 'net availability', Week '$w_{b.y}$'), c - x - d>*
2. · *<DC_001(Employee 'a', Fact 'net availability', Week '$w_{b.y}$'), e - x + x'>*

Analytics

1. Input cells: 3, Updated cells: 1
2. Input cells: 3, Updated cells: 1

Update 2

Update: *<R_001(Employee 'a', Week $w_{b.y}$), Team 't'>* This update is applied to relation *001 Employee x Week - Team* and changes the cell at Employee a and Week $w_{b.y}$ to Team t.

For the remainder of this update:

- Let t'_i be the previous value of *R_001(Employee 'a', Week 'i')*
- Let D'_i be the department to which t'_i belongs
- Let $future\_weeks$ be $\{\ w_{i.j} \mid (j = y \land b+1 \le i \le len(y)) \lor (j = y+1 \land 1 \le i \le len(y+1))\}$
- Let $future\_weeks^+$ be $future\_weeks \cup w_{b.y}$

Figure 3.7: Statistics on update 1

sub-model ID	**# updated cells**	**# input cells method 1**	**# input cells method 2**
2	1	3	3
3	2	2 * len(y)	4
4	2	2	3
5	2	2 * $\|set(t)\|$	4
6	4	2 * ($\|set(D)\|$ + $\|set(T)\|$)	4
7	6	8	10
8	30	54	32
9	6	6 * len(y)	8
10	6	6 * len(y)	14
11	15	30	32

5 - Sum hours over employees to team (per week)
Definitions

- Let $c_{i,f,j,k}$ be *DC_003(Employee 'i', Fact 'f', Team 'j', Week 'k')*
- Let $d_{f,i,j}$ be the previous value of *DC_002(Fact 'f', Team 'i', Week 'j')*
- Let $e_{f,i}$ *DC_001(Employee 'a', Fact 'f', Week 'i')*

Update methods
For fact $f \in \{$*gross availability, holiday, education, net availability*$\}$, week $i \in future\_weeks^+$:

1. · *<DC_002(Fact 'f', Team 't', Week 'i'),* $\sum_{j \in set(t)} c_{j,f,t,i}$ >
   · *<DC_002(Fact 'f', Team 't'_i', Week 'i'),* $\sum_{j \in set(t'_i)} c_{j,f,t'_i,i}$ >
2. · *<DC_002(Fact 'f', Team 't', Week 'i'),* $d_{f,t,i} + e_{f,i}$ >
   · *<DC_002(Fact 'f', Team 't'_i', Week 'i'),* $d_{f,t'_i,i}$ - $e_{f,i}$ >

Analytics

1. Input cells: 4 * $\sum_{i \in future\_weeks^+} (\sum_{j \in set(t)} (1) + \sum_{j \in set(t'_i)} (1)) \approx 8 * |future\_weeks^+| * |set(t)|$,
   Updated cells: 8 * $|future\_weeks^+|$
2. Input cells: 12 * $|future\_weeks^+|$, Updated cells: 8 * $|future\_weeks^+|$

3.2.4 Discussion

Update 1

For this model it was straighforward to come up with methods for propagating the update. By looking at the sub-models, it was not difficult to extract its functionality in terms of what the updated cells are, and what computations it would perform, using which input cells.

Figure 3.8: Statistics on update 2, fw = $|future\_weeks|$ and fw+ = $|future\_weeks^+|$

sub-model ID	**# updated cells**	**#input cells method 1**	**#input cells method 2**
autofill	fw	1	-
4	4 * fw+	8 * fw+	-
5	8 * fw+	8 * fw+ * $\|set(t)\|$	12 * fw+
6	8 * fw+	8 * fw+ * $\|set(D)\|$	12 * fw+
7	16 * fw+	20 * fw+	24 * fw+
8	16 * fw+	16 * fw+ + 64	20* fw+ + 16
9	32	16 * (len(y) + len(y+1))	48
10	32	32 + 16 * (len(y) + len(y+1))	32 + 16 * (len(y) + len(y+1))
11	15	30	-
12	2 * fw+	0	-
13	2 * fw+	fw+ * ($\|set(t)\|$ + $\|set(t')\|$)	2 * fw+
14	2 * fw+	fw+ * $\|set(t')\|$	-

An interesting result of this analysis is that for the 2 update methods there is not a method that outperforms the other in terms of number of input cells. Depending on the sub-model, minimizing the actual sub-model performs better than applying x on the output dataset of the sub-model and vice versa.

An early rule of thumb for this model seems to be that the second method is more efficient when a function computes a value over a range of dimension values, in the case of this model this function is often a summation. This is the case in a rollup and summation operation, as can be noticed by the difference in input cells in sub-models 2, 3, 6 and 9. The reason this happens is that the first method uses all values in the range of dimension values, where method 2 does not.

This also applies to the moving total operation from sub-model 8, though to a lesser extent, which is noticed by the smaller factor between the number of input cells between the methods. This is explainable by the fact that the range of dimension values is smaller; in the summation and rollup operations the entire range of dimension values was used, where here only a subrange is used.

Another observation is the effect of the moving total operator on the cells that need to be updated in the model. Before this operator was used, in the week dimension only updates needed to be performed to cells corresponding to week $w_{b.y}$ (and year y). Then sub-model 8 has the effect that cells corresponding in the range week $w_{b.y}$ untill $w_{b+4.y}$ in datacube 004 need to be updated. Moreover, submodel 11 which has this datacube as input, need to update the weeks in this longer range as well.

This leads to the suggestion that if, for a given dataset, an update affects a range of values for a dimension (and the same values for other dimensions), then sub-models that depend on this dataset also need to update the values in this range. Another place where this happens is at sub-model 2, which updates cells corresponding to value *net availability* for the Fact dimension in datacube 001. Then all datasets depending on this dataset need to update cells corresponding to *net availability* as well, next to the dimension value *holiday*, which was updated in the first place. This phenomenon is henceforth refered to as *dimension range propagation*.

Update 2

For this update again an attempt was made to find multiple update propagation methods for each sub-model. This was often succesfull; for each sub-model a method is possible where again the same operations are simulated that are performed in the actual sub-model, and an alternative method for most sub-models could be found as well.

However finding alternative update methods was less intuitive then for *Update 1*, since for this method, in contrast to *Update 1*, it was not the case that a single value was changed that was directly used by its successors. Rather, this update changes a value that is almost exclusively used as dimension value and therefore it can be noticed that values 'transfer' to this new dimension value. Concretely: in *Update 1* the alternative method always constitutes taking the previous value for the affected output dataset cells and adding and subtracting x and x' or vice versa, where for *Update 2*, the alternative method constitutes subtracting a set of values from cells selected with t'_i, and adding this set to cells selected with t.

Also in this model dimension range propagation is present; a clear example is that because of the *autofill* property of relation 001 for the Week dimension the range *future_weeks* is affected, this range is propagated through the model.

The differences in performance between method 1 and 2 are similar to those in *Update 1*; when the sub-model constitutes a sum or rollup, then method 2 often avoids the need for an iteration over the dimension, or dimension level that is being summed. For *Update 2* this does not hold for rollup operations over the Week dimension, since a large range of values in this dimension is changed.

Running time analysis

We will analyse the number of cells in terms of n, which is the number of employees in the dataset. Employees is chosen, since this is the only dimension which is not bounded by a constant, except for some other dimensions such as the number of teams, however these depend on the number of employees.

Thus the sizes of sets y, t, t', fw and $fw+$ are bounded by a constant. T and D grow linearly with the number of employees. When looking at the table this means that for both update methods it holds that the number of updated cells is always (bounded by a) constant. This holds also for the number of input cells for both method 1 and 2, except for sub model 6, which is linear in the number of employees.

3.3 Relational algebra and Assemble models

The relational model is the primary data storage model for data-processing applications. It has this primary position because of its simplicity and independence from underlying data structures [12].

A relational database consists of a collection of *tables* also called *relations*. For the remainder of this report we will continue using the term relation (not te be confused with a relation in Assemble), however imagening such a relation as a table is intuitive; namely, a relation consists of columns, normally refered to as *attributes*, but we will refer to these as *dimensions*, and rows which are refered to as *tuples*, which are single database records.

A relation is normally refered to by a *relation schema*, *a(dim1, dim2, ...)* where *a* is the name of the relation and *dim1, dim2, ...* are the dimensions.

Finally we introduce the notion of *relational algebra*, consisting of a set of operations that take one or two relations as input and produce a new relation as their result. Such an expression are also known as *queries*. Then operators in the relational algebra, relevant for our project are selection σ, projection π, (natural) join $\bowtie$, union $\cup$, cartesian product $\times$ and anti-join $\triangleright$.

3.3.1 Motivation

Relational algebra and its query notation is a universally used format in the database research field. It refers to a dataset as 'relation', hence for this section this definition of relation applies as well. Then the columns will still be refered to as 'dimensions', but the rows, which were often called 'cell', in a relation are called 'tuple'.

Relational algebra is universal in the sense that it serves as a common denominator for query languages in general; a query in an arbitrary query language can almost always be converted to a query in relational algebra, and vice versa. Moreover, in the literature either relational algebra or SQL is used, which can almost always be converted to each other as well.

Indeed DYN also uses relational algebra. Therefore in order to apply DYN translating availability to queries in the relational algebra format is very useful. Also it may help for identifying whether an algorithm is applicable to the Assemble submodels. Namely, when a submodel can be converted, but it requires an operator that is not supported by the algorithm, then it is clear that this may become a challenge.

3.3.2 Converting submodels to relational algebra

In order to derive the queries that correspond to the submodels, an analysis was performed on the submodels and datasets by first converting the datasets to relations, as defined by the relational algebra, and creating example relations of the input data with only a few tuples. Converting Assemble datasets to relations can be done by the following procedure. An arbitrary datacube *DCy x per datacube_dimensions* is converted to relation $y(\underline{datacube\_dimensions}, x)$ and Assemble relation *Ry source(s) - target* is converted to relation $y(\underline{source(s)}, target)$. All submodels have been converted to relations, after which examples of the relations have been composed that are such that the behaviour of a submodel can be demonstrated by stating what tuples the submodel would produce. This process also showed that at a couple of occasions additional relations needed to be created for deriving values that are not present in the input relations.

In order to illustrate this process, consider submodel:
2 - Compute net availability per employee per week
Then the output relation is: *DC001(employee, fact, week, hours)*
Example of relation:

Employee	**Fact**	**Week**	**Hours**
Rik	gross avail.	w12.2022	40
Rik	holiday	w12.2022	12
Rik	education	w12.2022	8

Then the following tuple should be inserted into *DC001*

Employee	**Fact**	**Week**	**Hours**
Rik	net avail.	w12.2022	20

Since value *net avail.* for dimension $fact$ is not in the input dataset, we need as additional input relation *Factnet(fact)* containing tuple *("net availability")*:

Fact
net avail.

The query then is:

$$\begin{aligned}
&DC001_2 \leftarrow DC001_1 \cup \\
&\pi_{a.employee,d.fact,a.week,hours \leftarrow (a.hours-b.hours-c.hours)} \\
&\quad (\rho_a(\sigma_{fact=grossavail.}(DC001_1)) \bowtie_{a.employee=b.employee \wedge a.week=b.week} \\
&\quad \rho_b(\sigma_{fact=holiday}(DC001_1)) \bowtie_{b.employee=c.employee \wedge b.week=c.week} \\
&\quad \rho_c(\sigma_{fact=education}(DC001_1)) \times \\
&\quad \rho_d(Factnet))
\end{aligned}$$

These kinds of examples for the other submodels of availability calculations can be found in the appendix.

The nafill operator

Note that the $nafill$ operator is not taken into consideration yet, where it does have to be taken care of at some point. One can take the net availability query as example. When considering:

$$\begin{aligned}
&(\rho_a(\sigma_{fact=grossavail.}(DC001_1)) \bowtie_{a.employee=b.employee \wedge a.week=b.week} \\
&\ \rho_b(\sigma_{fact=holiday}(DC001_1)) \bowtie_{b.employee=c.employee \wedge b.week=c.week} \\
&\ \rho_c(\sigma_{fact=education}(DC001_1)) \times \\
&\ \rho_d(Factnet))
\end{aligned}$$

Also note that a tuple for an employee and week will, unwillingly, not be generated when one or two of tuples with this *employee* and *week* and *fact*: *holiday* or *education* is not available.

One strategy to solve this problem is by altering the relation beforehand by inserting missing tuples according to the $nafill$ function, hence in this case by inserting tuples with $Hours$ value 0, when these are not available. In this example this is viable. This is because for each *employee* and *week* combination a tuple *net avail.* needs to be computed, hence adding for each such combination two other tuples only increases the number of added tuples linearly in the number of *net avail.* tuples.

The alternative is to alter the query by checking for na values implicitely. For example the following query computes for the example submodel (assuming that when there is an *employee*, then there is a a tuple for each $week$ and $fact$ $grossavail.$):

$$\begin{aligned}
&HolidayNA' \leftarrow \pi_{employee,week}(DC001_1) - \pi_{employee,week}\sigma_{fact=\text{"}holiday\text{"}}(DC001_1) \\
&EducationNA' \leftarrow \pi_{employee,week}(DC001_1) - \pi_{employee,week}\sigma_{fact=\text{"}education\text{"}}(DC001_1) \\
&BothNA \leftarrow HolidayNA' \cap EducationNA' \\
&HolidayNA \leftarrow HolidayNA' - BothNA \\
&EducationNA \leftarrow EducationNA' - BothNA
\end{aligned}$$

$$DC001_2 \leftarrow DC001_1 \cup$$
$$\pi_{a.employee,d.fact,a.week,hours \leftarrow (a.hours - b.hours - c.hours)}$$
$$(\rho_a(\sigma_{fact=grossavail.}(DC001_1)) \bowtie_{a.employee=b.employee \wedge a.week=b.week}$$
$$\rho_b(\sigma_{fact=holiday}(DC001_1)) \bowtie_{b.employee=c.employee \wedge b.week=c.week}$$
$$\rho_c(\sigma_{fact=education}(DC001_1)) \times$$
$$\rho_d(Factnet))$$
$$\cup$$
$$\pi_{a.employee,c.fact,a.week,hours \leftarrow (a.hours - b.hours)}$$
$$(\rho_a(\sigma_{fact=grossavail.}(DC001_1 \bowtie HolidayNA)) \bowtie_{a.employee=b.employee \wedge a.week=b.week}$$
$$\rho_b(\sigma_{fact=education}(DC001_1 \bowtie HolidayNA)) \times$$
$$\rho_c(Factnet))$$
$$\cup$$
$$\pi_{a.employee,c.fact,a.week,hours \leftarrow (a.hours - b.hours)}$$
$$(\rho_a(\sigma_{fact=grossavail.}(DC001_1 \bowtie EducationNA)) \bowtie_{a.employee=b.employee \wedge a.week=b.week}$$
$$\rho_b(\sigma_{fact=holiday}(DC001_1 \bowtie EducationNA)) \times$$
$$\rho_c(Factnet))$$
$$\cup$$
$$\pi_{employee,b.fact,week,hours}$$
$$(\rho_a(\sigma_{fact=grossavail.}(DC001_1 \bowtie BothNA)) \times$$
$$\rho_b(Factnet))$$

Although possible, rewriting each query - belonging to a submodel with one or more *nafill* operators - does require some work and also makes these a lot less concise, where each combination of missing *na* values requires its own relation that needs to be unified with the output relation. Since solving the '*na* problem' may be solved in different ways, for now it will be disregarded, but note that for a complete end product this needs to be taken into account.

Describe output datasets in terms of input sets only

It may be interesting to examine whether the output datasets can be expressed by a query in terms of input datasets only, hence without defining queries over sub-results. As example take $DC001_3$. Then when having the following input relations:
$DC001_0$

Employee	**Fact**	**Week**	**Hours**
Rik	holiday	w12.2022	12
Rik	education	w12.2022	8
Rik	holiday	w13.2022	0
Rik	education	w13.2022	20
Anne	holiday	w12.2023	0
Anne	education	w12.2023	0

$DC005$

Employee	**Week**	**gross availability**
Rik	w12.2022	40
Rik	w13.2022	40
Anne	w12.2023	30

$Factgross$

Fact
gross avail.

Factnet

Fact
net avail.

Then the query should produce:

Employee	**Fact**	**Week**	**Hours**
Rik	gross avail.	w12.2022	40
Rik	holiday	w12.2022	12
Rik	education	w12.2022	8
Rik	net avail.	w12.2022	20
Rik	gross avail.	w13.2022	40
Rik	holiday	w13.2022	0
Rik	education	w13.2022	20
Rik	net avail.	w13.2022	20
Anne	gross avail.	w12.2023	30
Anne	holiday	w12.2023	0
Anne	education	w12.2023	0
Anne	net avail.	w12.2023	30
Rik	gross avail.	2022	80
Rik	holiday	2022	12
Rik	education	2022	28
Rik	net avail.	2022	40
Anne	gross avail.	2023	30
Anne	holiday	2023	0
Anne	education	2023	0
Anne	net avail.	2023	30

First arrises the question whether this relation can be produced from a single query, hence not a union of multiple queries. This is not possible. One of the reasons why this does not work is that the Hours dimension is computed in a different manner for tuples with Fact 'net availability' and Fact 'holiday' (both having a Week value on the week level). Hence unifying multiple queries is unavoidable.

Then when attempting to create this result with a unification of queries, it is possible to produce all tuples with a Week value on the week level, without using sub-results:

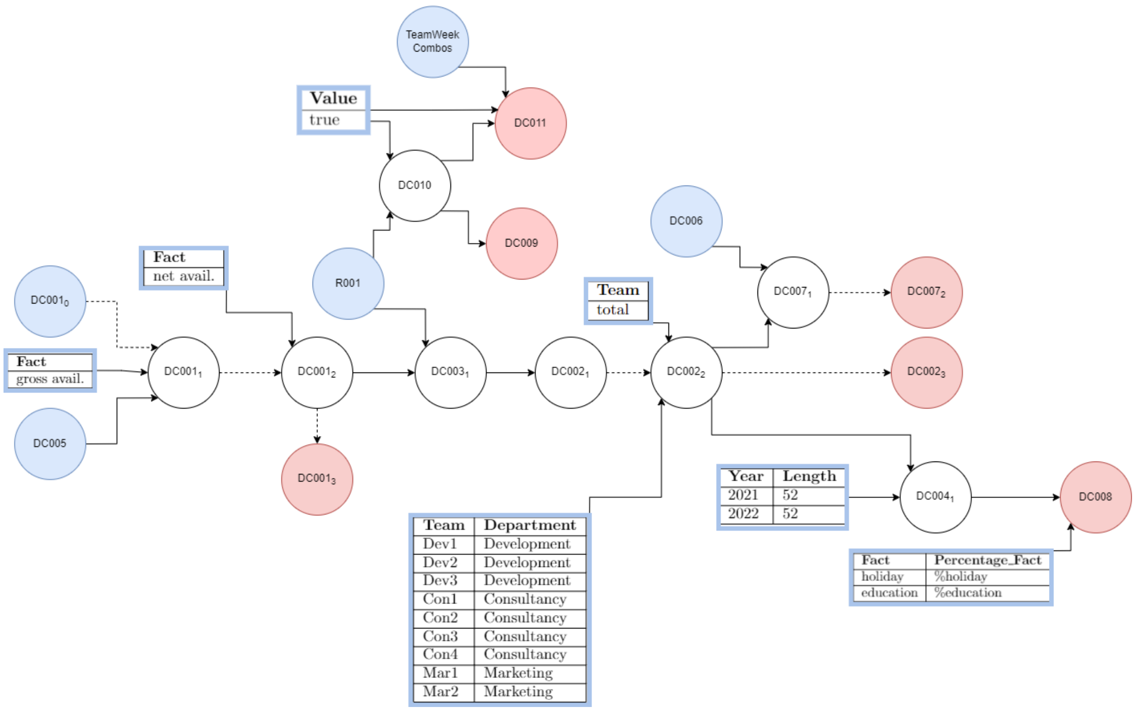
$$
\begin{aligned}
&DC001_0 \cup \\
&\pi_{employee,fact,week,hours}(DC005 \times Factgross) \cup \\
&\pi_{a.employee,d.fact,a.week,hours \leftarrow (a.hours - b.hours - c.hours)} \\
&\quad \sigma_{b.fact=holiday \wedge c.fact=education} \\
&\qquad (\rho_a(DC005) \bowtie_{a.employee=b.employee \wedge a.week=b.week} \\
&\qquad \rho_b(DC001_0) \bowtie_{b.employee=c.employee \wedge b.week=c.week} \\
&\qquad \rho_c(DC001_0) \times \\
&\qquad \rho_d(Factnet))
\end{aligned}
$$

However, at this point it can be noticed that for computing the tuples with a Week value on the year level, the query does explicitly need tuples produced by the previous results. Then these either need to be computed again, or the previous results can be reused. The latter seems to be favorable; in terms of time management computing the same tuples twice cannot be more efficient than only doing this once and since the previous result, assuming that these will not be duplicated, is contained by the output dataset, computing the tuples again is also not more efficient in terms of space management.

Total query

The complete list of all queries in availability calculations, one for each submodel is given below. Note that query results, which are relations themselves, are used by other queries. Hence the queries together form a DAG $G = (V, E)$, where $v \in V$ represents a relation, (which is sometimes the result of a query), and $e = (v1, v2) \in E \iff$ relation $v1$ is used in the query derivation of $v2$. Figure 3.9 shows this DAG. Where blue nodes are input relations only and red nodes are output relations only.

Figure 3.9: Query respresentation of availability calculations



Value
true

Fact
net avail.

Fact
gross avail.

Team
total

Team	**Department**
Dev1	Development
Dev2	Development
Dev3	Development
Con1	Consultancy
Con2	Consultancy
Con3	Consultancy
Con4	Consultancy
Mar1	Marketing
Mar2	Marketing

Year	**Length**
2021	52
2022	52

Fact	**Percentage_Fact**
holiday	%holiday
education	%education

1. $DC001_1 \leftarrow DC001_0 \cup \pi_{employee,fact,week,hours \leftarrow grossavail.}(DC005 \times Factgross)$

2. $$\begin{aligned} & DC001_2 \leftarrow DC001_1 \cup \\ & \pi_{a.employee,d.fact,a.week,hours \leftarrow (a.hours - b.hours - c.hours)} \\ & \quad (\rho_a(\sigma_{fact=grossavail.}(DC001_1)) \bowtie_{a.employee=b.employee \wedge a.week=b.week} \\ & \quad \rho_b(\sigma_{fact=holiday}(DC001_1)) \bowtie_{b.employee=c.employee \wedge b.week=c.week} \\ & \quad \rho_c(\sigma_{education}(DC001_1)) \times \\ & \quad \rho_d(Factnet)) \end{aligned}$$

3. $DC001_3 \leftarrow DC001_2 \cup \pi_{employee,week \leftarrow week.y,fact,hours \leftarrow (\mathbf{sum}(hours))}(DC001_2)$

4. $DC003_1 \leftarrow \pi_{employee,fact,team,week,hours}(DC001_2 \bowtie R001)$

5. $DC002_1 \leftarrow \pi_{fact,team,week,hours \leftarrow (\mathbf{sum}(hours))}(DC003_1)$

6. $DC002'_2 \leftarrow \pi_{fact,team \leftarrow department,week,hours \leftarrow (\mathbf{sum}(hours))}(DC002_1 \bowtie TeamDepartment)$
   $DC002_2 \leftarrow DC002_1 \cup DC002'_2 \cup \pi_{fact,b.team,week,hours \leftarrow (\mathbf{sum}(hours))}(DC002'_2 \times \rho_b(Teamtotal))$

7. $DC007_1 \leftarrow \pi_{fact,team,week,adjusted\_hours \leftarrow hours*(100+percentage)/100}(DC002_2 \bowtie DC006)$

8. $002withYL \leftarrow \pi_{fact,team,week,hours,length}(DC002_2 \bowtie_{DC002_2.week.y=Yearlengths.year} Yearlengths)$
   $\theta \leftarrow a.fact = b.fact \wedge$
   $a.team = b.team \wedge$
   $((a.week.y = b.week.y \wedge a.week.w - b.week.w \geq 0 \wedge a.week.w - b.week.w < 5) \vee$
   $(a.week.y = b.week.y + 1 \wedge a.week.w + b.length - b.week.w \geq 0 \wedge a.week.w + b.length -$
   $b.week.w < 5))$

   $DC004_1 \leftarrow \pi_{a.fact,a.team,a.week,hours \leftarrow (\mathbf{sum}(b.hours))}(\rho_a(DC002) \bowtie_\theta \rho_b(002withYL))$

9. $DC002_3 \leftarrow DC002_2 \cup \pi_{fact,team,week \leftarrow week.y,hours \leftarrow (\mathbf{sum}(hours))}(DC002_2)$

10. $DC007_2 \leftarrow DC007_1 \cup \pi_{fact,team,week \leftarrow week.y,adjusted\_hours \leftarrow (\mathbf{sum}(hours))}(DC007_1)$

11. $DC008 \leftarrow$
    $\pi_{a.percentage\_fact,a.team,a.week,percentage \leftarrow (a.hours/b.hours)}($
    $\rho_a((\sigma_{fact==\text{"}holiday\text{"}}(DC004_1) \bowtie PFHoliday) \cup (\sigma_{fact==\text{"}education\text{"}}(DC004_1) \bowtie PFEducation))$
    $\bowtie_{a.team=b.team,a.week=b.week}$
    $\rho_b(\sigma_{fact=\text{"}grossavailability\text{"}}(DC004_1))$
    $)$

12. $DC010 \leftarrow R001 \times Valuetrue$

13. $DC009 \leftarrow \pi_{team,week,count \leftarrow (\mathbf{count}(employee))}(DC010)$

14. $DC011 \leftarrow (TeamWeekCombos \rhd \pi_{team,week}(DC010)) \times Valuetrue$

Discussion

As can be seen, each submodel has an equivalent query. The join operators that are used are the cartesian product, natural join and join with a predicate θ. Many queries make use of relation aliasing, and the set operators union and minus were used. Intersection was not used, but may be still used and division was not used and is likely also incompatible with the DYN algorithm.

Four extensions needed to be made to the relational algebra that are not standard. These extensions are to assign to a projected dimension:

1. Algebraic expressions

2. Aggregation functions. Herefore the notation $\mathcal{G}_{GB}\mathbf{F}(x)$ is used, with $\mathbf{F}$ being the aggregation function, x the aggregated dimension and $\mathcal{G}_{GB}$ the group-by expression, consisting of group-by operator $\mathcal{G}$ and comma separated list GB of group-by dimensions. However, we omit the group-by expression as GB is always equal to the projected dimensions.

Chapter 4

The current state of Dynamic Yannakakis

4.1 Overview

In this chapter a summary and discussion of the components and concepts of DYN will be given. Doing this, the goal is to be able to answer our first research question: 'What does Dynamic Yannakakis in its current state offer?'. Therefore we will look at what the algorithm offers in terms of functionality. Also the benefits of the algorithm will be discussed, both in terms of memory consumption and time complexity, where additionally a look will be given to whether or not these benefits will apply to our use case. Then at the end we discuss what is still missing in terms of functionality and will thus be needed to be accommodated for.

4.2 Datastructures and procedures

First we give an overview of the main datastructures and procedures concerning DYN. For now we will describe these in the way they are given in the papers by Idris et al.

DYN uses two main datastructures. The first one is a *General Join Tree* (GJT). A GJT T is a tree (V, E), thus consisting of nodes and edges, which follows a query plan such that it will be able to calculate the result of a query Q. The second datastructure is the T-reduct. This is an extension of a GJT, where each node n in the GJT is augmented with a relation ρ_n. As the T-reduct contains all information of the GJT, mainly the T-reduct will be discussed from now.

Then there are two procedures that together form the DYN algorithm. The first procedure is the *update* procedure which runs when one of the input relations of Q has been changed. The update procedure then accomplishes that in a bottom-up manner, the updates are propagated. As a result each $n \in V$, ρ_n will be updated appriopiately. How this exactly works will be discussed later.

The second procedure is the *enumeration* procedure. This runs on demand and will output the result of Q. This procedure consists of recursive calls of an algorithm $enum(n, t)$, for a node n and tuple t. It starts in the root of T and for each node n it calls $enum(c, t_c)$ on the children c of n and tuples $t_c \in \rho_c$.

4.3 T-reduct

Now we take a look at T-reducts in more detail. Let $out(Q)$ be the dimensions that are in the outer projection statement of Q. Each node n has a set of variables $var(n)$, where each variable

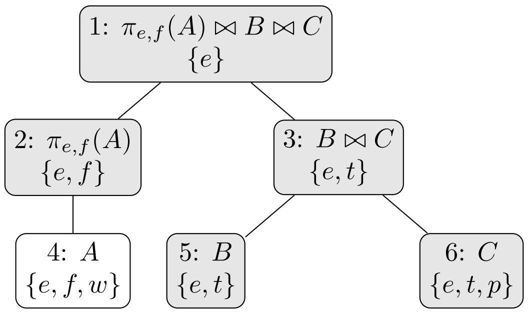


Figure 4.1: GJT for query $\pi_{e,f,t,p}$ *(A(e, f, w)* $\bowtie$ *B(e, t)* $\bowtie$ *C(e, t, p)*

in var(n) is associated to one of the dimensions in ρ_n.

A certain subset of V is called the frontier, denoted as F. Consider subset N of V such that $n \in N \iff n \in F \vee n$ is an ancestor of a node in F. Then F is chosen such that $n \in F \iff var(n) \subseteq out(Q)$. Moreover we have that $out(Q) = var(N)$

F is chosen as it is, in order to facilitate an efficient enumation procedure. Namely, as explained in this procedure an Enum() function is recursively called on the children of a node n. However Enum() will not be called from a node n if $n \in F$. In this way, as we now know that if a node n has been visited, and since this means that $var(n) \subseteq out(Q)$, the procedure only has to consider values from dimensions that need to be in the result of Q.

Some other properties hold for nodes in the frontier: if $n \in N$, then if n has a sibling m, this implies $m \in N$. This defined as the sibling-closed property. Also if a node n is in F then there is no node which is an ancestor or descendant of n which is also in F.

It is also interesting to look at the role of the individual nodes. The leaves of T correspond 1-to-1 to the input relations of Q. An inner node n in T is as follows:

1. n has one child c: $var(n) \subset var(c)$ and the edge (c, n) denotes an aggregation step over dimension set $s = var(c) \setminus var(n)$.

   This step is mostly used when $n \in F$, such that the enumeration procedure only has to consider n and not c. This is desirable as we know that, from the way F is constructed, if a dimension $q \in s$, then $q \notin out(Q)$.

   Also this aggregation step can be used if the query explicitely states that an aggregate needs to be computed, however the examples of the paper do not state such queries. In contrary, the Assemble queries may contain such an explicit aggregation step.

2. n has two children $c1$ and $c2$. Then n denotes a join step, and thus to an explicit join operator in Q. In this case it holds that either $c1$ or $c2$ we can denote with x, such that it holds that $var(n) \subseteq var(x)$.

As example, consider query $Q = \pi_{e,f,t,p}$ *(A(e, f, w)* $\bowtie$ *B(e, t)* $\bowtie$ *C(e, t, p)*. Figure 4.1 shows GJT T corresponding to Q. Each node n in T is labeled by an integer $i, 1 \leq i \leq 6$. Also it shows the query operation it corresponds to, or the relation name if v corresponds to an atomic relation. Finally it shows $var(n)$.

Then $out(T) = \{e, f, t, p\}$ and $N = \{1, 2, 3, 5, 6\}$ which can be observed by the fact that only node 4 does not have a grey color. Thus $F = \{2, 5, 6\}$. Also notice that leaf nodes $4, 5$ and 6 correspond to atomic relations A, B and C respectively and inner nodes 1 and 3 correspond to join operators in Q. Node 2 does not explicitly correspond to an operator in Q, but is necessary to ensure that $out(Q) = var(N)$ and T is sibling-closed.

4.4 Update procedure

Now we look at the update procedure in more detail. The procedure is described by algorithm 1. Observe that with each node n a relation Δ_n is associated, which describes the tuples to be added

Algorithm 1 Update procedure

```
1: procedure UPDATE($u$)
2:     for each $n \in$ leaves($T$) labeled by $r(\overline{x})$ do
3:         $\Delta_n \leftarrow u_{r(\overline{x})}$
4:     for each $n \in V \setminus$ leaves($T$) do
5:         $\Delta_n \leftarrow$ empty relation over $var(n)$
6:     for each $n \in V$ do, traversed bottom-up
7:         $\rho_n$ += $\Delta_n$
8:         if $n$ has a parent $p$ and a sibling $m$ then
9:             $\Delta_p$ += $\pi_{var(p)}(\rho_m \bowtie \Delta_n)$
10:        else if $n$ has parent $p$ then
11:            $\Delta_p$ += $\pi_{var(p)}\Delta_n$
```

and removed from ρ_n. u holds the updates to the input relations, hence to the leaves in T.

Thus in line 3, if for to a leaf l, labeled by relation $r(\overline{x})$, an update needs to be made, then this update is in u and Δ_l will be set to $u_{r(\overline{x})}$.

In line 5 for all remaining nodes n, Δ_n will be set to an empty relation.

In the second step of the procedure, in a bottom-up manner for each node n, first in line 9 the update will be applied to ρ_n and afterwards the delta set of the parent is computed.

Again there is a distinction for when a node has one child or when a node has two children. As we know when a node has two children, this corresponds to a join operation and in the other case to an aggregation. In the latter case, although it is not very explicit, values necessary for computing aggregates, namely *count* of the amount of tuples and *sum* of the aggregated value, are maintained. For the former case also the *count* will be maintained, such that when removing tuples from some ρ_p, we know when for a tuple t in ρ_p there are no longer tuples associated in ρ_c for any child c of p, and therefore t can be removed from ρ_p. Otherwise the count of t will simply be decremented.

Example

Figure 4.2 shows a T-reduct T before and after an update u has been processed. First note that each node in the tree has been augmented with a relation, which is the ρ relation. Then for each tuple there is an extra attribute #, which is not part of the tuple but a piece of information used for maintaining the tree. More specifically, # denotes the *multiplicity* for a tuple $t \in \rho_n$ for some node n, which tells how many other tuples in descendants of n are associated with t, such that it is known when a tuple is associated with 0 tuples and therefore can be removed from the ρ relation. For a tuple in a leaf node, # is always 1.

Now we consider the update. Observe that to node B a tuple with $week = W12.2022$ has been added, and from node C a tuple with $week = W11.2022$ has been removed. As consequence A has two tuples with $week = W12.2022$, such that the multiplicity of tuple [$W11.2022$] in ρ_1 is $2 * 1 = 2$. Additionally B does not have any tuples with $week = W11.2022$ anymore, thus the multiplicity of tuple [$W12.2022$] in ρ_1 is $1 * 0 = 0$ and the tuple is removed from ρ_1.

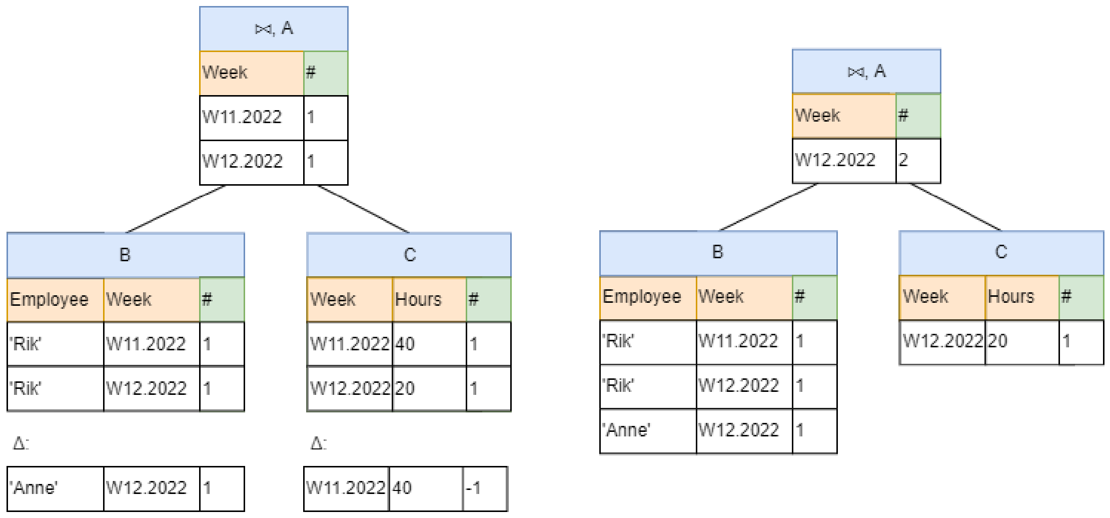


Figure 4.2: Update procedure for a tree with a join node as root: before and after.

4.5 Enumeration procedure

Finally we consider the enumeration procedure. This procedure is described by algorithm 2 for T-reduct T. As described, the enumeration procedure in the paper consists of an initial call Enum()

Algorithm 2 Enumeration procedure

```
1:  function ENUM
2:      for each $t \in \rho_{root_T}$ do
3:          Enum($root_T$, $t$)
4:
5:  function ENUM($n$, $t$)
6:      if $n \in F$ then
7:          yield $t$
8:      else if $n$ has one child $c$ then
9:          for each $t1 \in \rho_c \ltimes t$ do
10:             Enum($c$, $t1$)
11:     else
12:         for each $t1 \in \rho_{c1} \ltimes t$ do
13:             for each $t2 \in \rho_{c2} \ltimes t$ do
14:                 for each $s1 \in$ Enum($c1$, $t1$) do
15:                     for each $s2 \in$ Enum($c2$, $t2$) do
16:                         yield ($s1 \cup s2$)
```

In this function, a call $Enum(root_T, t)$ will be made for each tuple $t \in \rho_{root_T}$. This puts in motion a series of recursive calls Enum(n, t) for any arbitrary node $n \in N$ and tuple t. In the Enum(n, t) function there are three cases.

First, if $n \in F$ we simply return t.

Else, if n has one child c, the algorithm first find all tuples $t1$ in ρ_c that are compatible with t and then makes a recursive call Enum() with $t1$ and c as parameters. The result of this call will then be returned.

Finally, if n has two children $c1$ and $c2$, then first all tuples $t1$ and $t2$ in ρ_{c1} and ρ_{c2} respectively that are compatible with t are found, after which the Enum() function will be recursively called on both $c1$ and $t1$, and $c2$ and $t2$. The results of these calls $s1$ and $s2$ will be unified, which then will be yielded.

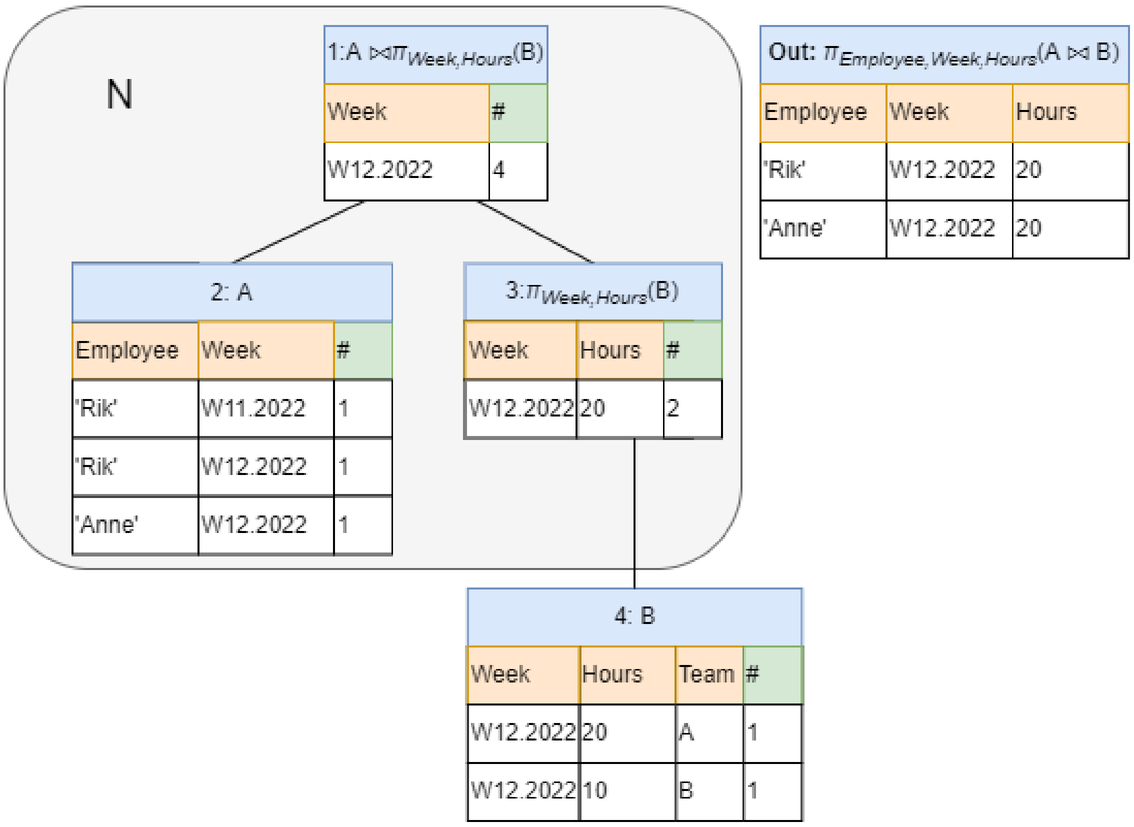
Note that in Enum(), all tuples in ρ_{root_T} will be iterated over. Now if T is according to Q, then given that T is up-to-date, calling Enum() will return all output tuples of Q.

Example

As example consider figure 4.3. It displays a GJT T and the resulting relation for executing the enumeration procedure. Observe that although the relation denoted by node 1 and the output relation on first glance do not seem equal, but actually are.

In this case it was not possible to first join nodes 2 and 4 and afterwards have an aggregation node as root, such that the relation expression would be exactly the same, since then the $out(Q) = var(N)$ and sibling-closed property combination could not be both possible for T.

Regarding the enumeration procedure, the algorithm iterates over tuple $[W12.2022]$ in ρ_1. This tuple is compatible with $S1 = \{\ ['Rik', W12.2022], ['Anne', W12.2022]\} \subset \rho_2$ and tuple $[W12.2022, 20] \in \rho_3$. Then tuple $s1 \cup [W12.2022, 20]$ is returned for each $s1 \in S1$.



1:A $\ltimes \pi_{Week,Hours}$(B)

Week	#
W12.2022	4

2: A

Employee	Week	#
'Rik'	W11.2022	1
'Rik'	W12.2022	1
'Anne'	W12.2022	1

3:$\pi_{Week,Hours}$(B)

Week	Hours	#
W12.2022	20	2

4: B

Week	Hours	Team	#
W12.2022	20	A	1
W12.2022	10	B	1

Out: $\pi_{Employee,Week,Hours}$(A $\bowtie$ B)

Employee	Week	Hours
'Rik'	W12.2022	20
'Anne'	W12.2022	20

Figure 4.3: A GJT and its enumeration result.

4.6 Complexity of DYN

As described by the paper, DYN is designed to be efficient both in terms of memory and time complexity.

The efficiency in terms of memory complexity stems from the fact that DYN offers an alternative to IVM, mainly by not requiring the user to materialize the output relation and also subresults, such that the space required by DYN is $\mathcal{O}(|db|)$ opposed to $\mathcal{O}(|Q(db)|)$, db being the input database. This can be bad if output $Q(db)$ is substantially larger than input db.

Concerning time complexity, there are two cases:

If T has an edge of which the predicate has multiple inequalities, then the update procedure of DYN takes $\mathcal{O}(M * \log(M))$ time, where $M = (|db| + |u|)$ and the enumeration procedure enumerates with constant delay, meaning that the time complexity of $Enum()$ is $\mathcal{O}(Q(|M|))$.

Else, thus if there is at most one inequality in each predicate of Q, then the update procedure takes $\mathcal{O}(M^2 * \log(M))$ time, and then the enumeration procedure enumerates with a delay of $\mathcal{O}(log(|M|))$, resulting in a time complexity for $Enum()$ of $\mathcal{O}(Q(|M|) * log(|M|))$

4.7 Cyclic queries

For a query Q to be suitable for DYN, Q has to be acyclic. For example, the triangle query $A(x,y) \bowtie B(y,z) \bowtie C(x,z)$ is cyclic and therefore not suitable for DYN. The reason is that for such a query there does not exist a GJT T, such that:

- Each inner node $n \in T$ has child c such that $var(n) \subseteq (var(c))$.
- For each variable $v \in var(T)$ it holds that if $v \in var(n1) \wedge v \in var(n2)$ for two arbitary nodes $n1, n2 \in T$, then $v \in var(n3)$ for any $n3 \in (n1, n2)$, where $(n1, n2)$ is the path from $n1$ to $n2$.

Which are properties that both should hold for a GJT.

The AC model does contain such a cyclic query. In section 5.4 we show how this query is split into two sub-queries, such that these sub-queries are not cyclic and therefore the output relation of this query can still be maintained. This is achieved by letting the output relation of the first sub-query's GJT be an input relation of the second sub-query's GJT.

Indeed the solution for maintaining output relations of cyclic queries in general, is to split them into acyclic sub-queries, where each sub-query is converted to one GJT.

4.8 Conclusion

Concluding, we now are able to answer what DYN does offer in its current state. In order to know how we have to extend DYN, we also discuss what DYN does not offer yet in terms of functionality.

The algorithms described by the paper offer a way of constructing and dynamically maintaining a data structure in $\mathcal{O}(|db|)$ space and $\mathcal{O}(M * \log(M))$ or $\mathcal{O}(M^2 * \log(M))$ time, such that Q can be evaluated by enumerating the data structure with constant delay, or logarithmic delay, discriminating the cases where Q does not have a predicate with more than one inequality or it does, respectively.

Q can be any conjunctive query, thus composed of selection, projection and join statements. Also Q may contain aggregation functions.

4.8.1 What does DYN not offer?

DYN supports T-reduct based on queries with projection, join operators over a predicate. It also considers aggregations. Although the algorithm is not very explicit about maintaining values that are important for computing aggregations, these values being *sum* and *count*, it does so in an abstract manner, such that I do not deem it necessary to extend the already stated algorithms for aggregation nodes.

There are also query operators which are not (adequately) considered, whereas they are part of the queries derived from the Assemble submodels:

- Selection: the paper mentions the use of a selection operator but only shows how to simplify in the case where the selection is performed over the join of two relations, thus it shows that $\sigma_\theta(A \bowtie_\gamma B)$, can be simplified to $A \bowtie_{\theta \wedge \gamma} B$; the case where there is a selection over only one atomic relation, thus $\sigma_\theta A$ which cannot be simplified, is not explicitely considered.

- Union: although the procedures facilitating this operator will be fairly trivial, they are not provided by the paper.

- Aggregation over join: when a query contains both an aggregation function and a join operator, one may think that a T-reduct with the right query plan can be constructed in the following manner: the parent of the relations to be joined is a join node and that node's parent is an aggregate node. However, this will not work since the aggregated value has to be maintained, which is not done in the join node. For our queries we need only an aggregation join over two relations.

- Anti-join: corresponding to the *not any* statement in Assemble, the paper does not facilitate algorithms for queries containing relations A and B, for which a tuple in A may not join with any tuple in B. For our queries we only need to consider the case where $var(A) \subseteq var(b)$.

Next to these query operators, although the existence of enumeration procedures over delta nodes is surely mentioned and used, the procedure is merely explained as being a slightly modified version of the regular enumeration procedure. This may be true, however as these modifications necessary for realizing delta enumeration procedures may be not so trivial, they will be considered explicitly for our implementation.

Finally we should construct a procedure for updating all output relations corresponding to the output relations of AC. As this model consists of submodels with output datasets, which then are used as input datasets for other submodels, we should describe how an update would propagate through such a connected series of T-reductions. In this way we are able to describe a method which truely facilitates IVM for AC.

Chapter 5

Extending Dynamic Yannakakis

5.1 Chapter overview

In this chapter we take a look at how the DYN procedures from the paper will be extended, such that this extended algorithm facilitates the simulation of AC and thus IVM for the AHA output datasets.

In the first section we consider general alterations, we introduce the notion of an *output relation* for an enumeration function, we introduce the concept of *delta enumeration* and give a general overview of the IVM method that we implemented.

Secondly we consider the introduced node types and their corresponding algorithms for computing delta relations and output relations via algorithm *ComputeDelta* and (delta) enumeration algorithms respectively.

Thirdly we look at how we converted the queries corresponding to AC were converted to T-reductions.

Finally we give an evaluation for our extension. We consider time complexity, a major drawback of the current implementation and feasibility of the current implementation regarding the actual demands of Anago. Also in this final section we give a conclusion in which we give (provisional) answers to the questions 'Can we extend Dynamic Yannakakis to facilitate IVM for Assemble models, without nullifying the benefits of the current Dynamic Yannakakis algorithm?' and 'Considering the way Assemble is intertwined with the Oracle OLAP DML, would Dynamic Yannakakis be a practical short-term solution?'.

5.2 General alterations and extensions

5.2.1 Reconsidering the update and enumeration functions

Update procedure

Consider algorithm 1. The overall update procedure will be the same, except for the case distinction in lines 8 up until 11. Assume that in line 6 we iterate over some node n with parent p. Then this case distinction will be replaced by a call $p.ComputeDelta(n)$. The alterion is shown in algorithm 3

The reason why ComputeDelta() is called on node p is that each node will have their own update procedure, depending on its node type. Thus ComputeDelta() is called on parent node p, such that the implementation of ComputeDelta() is the responsibility of p.

Also observe that node n is given as a parameter to the function. This is necessary because in line 6 nodes are enumerated over in an arbitrary order. Therefore, since at line 7 we have that update Δ_n is applied to ρ_n, when calling p.ComputeDelta(n) in this way we can let p know that n's update has been applied, such that ComputeDelta will act accordingly. We do not know, if n

has sibling m, whether ρ_m has been updated. However we do know that if Δ_m is not empty, then call p.ComputeDelta(m) is made after ρ_m has been updated.

Algorithm 3 Altered update procedure

```
1: procedure UPDATE($u$)
2:     for each $n \in$ leaves($T$) labeled by $r(\overline{x})$ do
3:         $\Delta_n \leftarrow u_{r(\overline{x})}$
4:     for each $n \in V\setminus$ leaves($T$) do
5:         $\Delta_n \leftarrow$ empty relation over $var(n)$
6:     for each $n \in V$ do, traversed bottom-up
7:         $\rho_n$ += $\Delta_n$
8:         if $n$ has a parent $p$ then
9:             $p$.ComputeDelta($n$)
```

Enumeration procedure

Now we go over the alterations that we made compared to algorithm 2. There are two key alterations made to the enumeration procedure. As an example the reader may consider algorithm 7, but these alterations apply to any enumeration procedure that is discussed going formward:

1. The node to which t belongs is no longer given as a parameter, additionally there is no case distinction determining what kind of node n is and acting accordingly. This change is made because each node is of a certain type with corresponding enumeration functions. As we later will see, these functions then will be called on the node, rather then having n as parameter.
2. The input parameter of the functions is no longer a tuple from n, but a tuple from the parent of n, which is indicated by the fact that this parameter is no longer called t, but t_p.

   The reason why this is done is because, as we will see, for the delta enumeration procedures it should be the responsibility of n to which tuples in Δ_n t_p is joined. Namely most of the time when adding tuples, t_p needs to be joined to Δ_n^+ and likewise when removing tuples, t_p needs to be joined with Δ_n^-, but this will not always be the case.

Regarding the second alteration, one may wonder what this means for $root_T$. Namely moving the semijoin in such a way implies that also $root_T$ has delta enumeration algorithms with input parameter t_p, but since t has no parent there is no t_p. What therefore is the case is that t_p for $root_T$ is always NIL. Then for each delta enumeration procedure we have that it iterates over $t, t \in S \ltimes t_p$, with S being some arbitrary relation. Then since $t_p = NIL$, we have that we iterate over elements in $S \ltimes NIL = S$, which is exactly what we should have.

5.2.2 Enumeration output

In order to be able to reason about the output of enumeration function calls, we look at what exactly the enumeration procedure should return. We discuss a concept that is not explicitly discussed in the paper, but which is very useful when reasoning about the correctness of the (delta) enumeration algorithms.

The idea behind the enumeration procedure is that it returns some output relation O, called on a T-reduct T which is according to a query expression Q. Now Q defines the steps to compute O, however technically we have that $Q = O$. This is because the steps in the expression also describes the relation itself. Now we will show that the enumeration procedure indeed returns Q and therefore O.

First, to give some intuition why this should be the case, note that a relation R can be recursively defined in the following manner:

$$\begin{aligned} R =& r \\ & |(R) \\ & |\kappa(R) \\ & |R\psi R \end{aligned}$$

With, as far as the queries from the AHA model are concerned, r being an atomic relation, $\kappa \in \{\pi, \sigma\}$ and $\psi \in \{\times, \bowtie, \cup, \triangleright\}$. Thus a relation is defined recursively in terms of other relations.

Then the query expression corresponding to a node in n in T can be recursively found following algorithm 4, in which the query expression for the output of n is constructed:

Algorithm 4 Find the query expressoin corresponding to a node

```
1: function FindR($n$): relation $R$
2:     if $n$ is a leaf then
3:         return $n.R$                          ▷ $n$ corresponds to an atomic relation $R$
4:     else if $n$ has one child $c$ then
5:         return $\kappa$(FindR($c$))           ▷ $n$ corresponds to an unary query operation $\kappa$
6:     else
7:         return FindR($c1$) $\psi$ FindR($c2$) ▷ $n$ corresponds to a binary query operation $\psi$
```

Definition 1 (P_n) An output relation P_n for a node n, $n \in T$ for some T-reduct T is a relation such that:

- P_n = FindR(n)
- $P_n \ltimes t = n$.Enum(t)

Thus, following definition 1 each node $n \in T$ has output relation P_n, which can be computed by running n.Enum(NIL).

Additionally following this definition since Q = FindR($root_T$) we have that $Q = P_{root_T}$. Then, since the initial call in the enumeration procedure is $root_T$.Enum(NIL) we have that it will return P_{root_T}, and therefore Q and O, as we claimed it would do.

5.2.3 Delta enumeration

Next to regular enumeration procedures, which produce output relations, we also need procedures for changes in output relations. The reason why we want this is as follows: although it is possible to compute P_{root_T} from scratch by running the regular enumeration procedure, it may be the case that $|Q(M)| >> |Q(\Delta_M)|$. In this case, given that $Q(\Delta_M)$ can be enumerated linearly in respect to $|\Delta_{root_T}|$ (which we will investigate in the evaluation and during experiments), we can speed up the enumeration procedure by a factor $k = \frac{|Q(M)|}{|Q(\Delta_M)|}$.

Although this concept of delta enumeration is used in the paper, it has not been explicitly stated, but merely mentioned as being a slightly modified version of the regular enumeration procedure. For completeness sake and in order to make sure that the alterations we made to the regular algorithm are clear, we take a look at the delta enumeration procedures.

Now we determine what the delta enumeration procedures should exactly return. Consider query Q, with T-reduct T, update u and an arbitrary node n. Then by definition 1, we have output relation P_n. Let P_n then be the relation after u has been applied and P_n' the relation before u

has been applied. We want to compute changes in P_n relative to P'_n. These need to be both the set of tuples that were added to P'_n and tuples that were removed from P'_n. Therefore we define relations $P_{\Delta_n^+} = P_n - P'_n$ and $P_{\Delta_n^-} = P'_n - P_n$ respectively. Then indeed we have that $t \in P_{\Delta_n^+} \Rightarrow t \in P_n \wedge t \notin P'_n$ and $t \in P_{\Delta_n^-} \Rightarrow t \in P'_n \wedge t \notin P_n$.

In order to enumerate over $P_{\Delta_n^+}$ and $P_{\Delta_n^-}$, we introduce $n.EnumAdded(t)$ which returns $\Delta^+_{P_n} \ltimes t$ and $n.EnumRemoved(t)$ which returns $\Delta^-_{P_n} \ltimes t$. Then as an enumeration procedure starts with a function call on $root_T$ with tuple NIL, the procedure will indeed enumerate over $\Delta^+_{P_{root_T}} \ltimes NIL$ and $\Delta^-_{P_{root_T}} \ltimes NIL$. Then, since $O = P_{root_T}$, we indeed enumerate over Δ^+_O and Δ^-_O.

When discussing the delta enumeration procedures for the several types of nodes, the goal is that an intuition is given why the delta enumeration functions indeed enumerate over the correct tuple sets. Although this intuition will look like (part of) a proof, we do realize that it is not a complete correctness proof. Especially often the intuition lacks the proof that any tuple that should be returned is also returned. However, by giving an intuition in this proof-like-structure we think it is the most convincing.

5.2.4 Order and nesting of delta enumeration algorithms

Consider a T-reduct T with output relation O. As we see in algorithm 7 the property is lost that each tuple t that the algorithm iterates over will be in Δ_O, as t will only be yielded at certain conditions. This will not only apply to nodes in the frontier, but for most node types. This has as consequence that the delta enumeration procedure does not enumerate with constant delay.

However the delta enumeration procedure does enumerate with constant delay over tuples t that may be in Δ_O, let's call such a tuple a *candidate*.

Therefore for a call n.EnumF(t), EnumF $\in$ {Enum, EnumAdded, EnumRemoved}, if we denote its running time as $\mathcal{T}(n.\text{EnumF}(t))$, then lemma 1 gives some properties on such a call.

Lemma 1 Running time of enumeration functions.

1. $\mathcal{T}(n.\text{Enum}(t)) = |P_n \ltimes t|$
2. $\mathcal{T}(n.\text{EnumAdded}(t)) \geq |\Delta^+_{P_n} \ltimes t|$
3. $\mathcal{T}(n.\text{EnumRemoved}(t)) \geq |\Delta^-_{P_n} \ltimes t|$

Hence, the delta enumeration functions may enumerate over more than only the output tuples. This we need to keep in mind while discussing the delta enumeration procedures. Namely it has as a consequence that it may be wise to put two loops that may have been nested next to each other, or invert the order of the loops in a nested loop. Result wise this will not make a difference, but concerning running time it may.

Now we give an intuition for the motivation of using a certain ordering in nested enumeration iterations or unnesting the enumeration calls all together. Here lemma 1 is used.

Consider algorithm 5. From lemma 1 we have that EnumExample1 has a runningtime of $\mathcal{T}(c1.\text{EnumAdded}(t)) + |P_{\Delta_n^+} \ltimes t|$ * $(|P_n \ltimes t| + |P_n \ltimes t|)$, whereas EnumExample2 has a runningtime of $|P_n \ltimes t| + |P_n \ltimes t|$ * $(\mathcal{T}(c1.\text{EnumAdded}(t)) + |P_{\Delta_n^+} \ltimes t|)$. This means that if $\mathcal{T}(n.\text{EnumAdded}(t)) >> |P_{\Delta_n^+} \ltimes t|$, then surely EnumExample1 is significantly faster and if they are equal then it does not matter.

Likewise compare EnumExample3 and EnumExample4. EnumExample3 has runningtime $\mathcal{T}(c1.\text{EnumAdded}(t)) + |P_{\Delta_n^+} \ltimes t|$ * $(\mathcal{T}(c2.\text{EnumRemoved}(t)) + |P_{\Delta_n^-} \ltimes t|)$, whereas EnumExample4 has runningtime $\mathcal{T}(c1.\text{EnumAdded}(t)) + \mathcal{T}(c2.\text{EnumRemoved}(t)) + (|P_{\Delta_n^+} \ltimes t|$ * $|P_{\Delta_n^-} \ltimes t|)$. Hence if either $\mathcal{T}(n.\text{EnumAdded}(t)) >> |P_{\Delta_n^+} \ltimes t|$ or $\mathcal{T}(n.\text{EnumRemoved}(t)) >> |P_{\Delta_n^-} \ltimes t|$, then EnumExample3 surely is significantly faster, where otherwise is does not matter much.

Algorithm 5 Enumerate examples

```
function EnumExample1($t$)
    for each $s1 \in c1.EnumAdded(t)$ do
        for each $s2 \in c2.Enum(t)$ do
            yield $s1 \cdot s2$

function EnumExample2($t$)
    for each $s2 \in c2.Enum(t)$ do
        for each $s1 \in c1.EnumAdded(t)$ do
            yield $s1 \cdot s2$

function EnumExample3($t$)
    for each $s1 \in c1.EnumAdded(t)$ do
        for each $s2 \in c2.EnumRemoved(t)$ do
            yield $s1 \cdot s2$

function EnumExample4($t$)
    $S1 \leftarrow c1.EnumAdded(t)$
    $S2 \leftarrow c2.EnumRemoved(t)$
    for each $s1 \in S1$ do
        for each $s2 \in S2$ do
            yield $s1 \cdot s2$
```

5.2.5 General update procedure

At this point we have gone over the general alterations made to the update and enumerate procedures and the concept of delta enumeration functions. Therefore this is a good moment to give a general overview of the output relation maintenance procedure.

Therefore consider algorithm 6, which is called on some T-reduct T. The idea is fairly straightforward.

In line 2 we first run the update procedure to apply updates on T. In this procedure first we have that for any relation ρ_l, leaf $l \in T$, if there is an update $u_r \in u$ such that $r = \rho_l$, then u_r is applied to ρ_l. Afterwards this update is propagated through T, to bring other sets ρ_n, Δ_n^+ and Δ_n^- up-to-date for any node $n \in T$.

Secondly, in lines 3 and 4 we run EnumAdded and EnumRemoved respectively, to compute the updates to O_T. Where in line 5 and 6 the added tuples are added to O_T and the removed tuples are removed from O_T.

Finally in lines 7 and 8 we add these updates to u, such that other T-reducts for which O is an input set are able to apply the updates made to O themselves.

Observe that we let each node apply updates to their own leaves. On first glance this may seem unnecessary. Namely, if for a T-reduct T' there is a leaf node $l' \in T'$ for which $\rho_{l'} = O_T$, then instead of applying the updates from u, which involves iterating over Δ_O, one may argue that we just do a constant time reasignment of ρ_l', by letting ρ_l' point to O_T.

From a relational point of view this is totally correct, however one should not forget that in this algorithm any relation corresponding to ρ_n, Δ_n^+, Δ_n^- or O_T, is implented by means of an index structure I_r, for relation r. Then such an index itself is a relation and implemented by means of a hash set. Each tuple/key in I_r points towards a list of tuples in r.

Then, for O_T, we have that there may be multiple relations $\rho_{l'}$ such that $O_T = \rho_{l'}$. As $var(I_{\rho_{l'}})$ depends on $var(p_{l'})$, where $p_{l'}$ is the parent of l' we have that I_{O_T} cannot be implemented such that $var(I_{O_T}) = var(I_{\rho_{l'}})$ for any such $\rho_{l'}$.

Hence each such $\rho_{l'}$ needs to go through all tuples in Δ_{O_T} and apply it to $I_{\rho_{l'}}$ itself.

Algorithm 6 Find the relation corresponding to a node

```
1: function UpdateTreeAndOutput($u$)
2:     Update($u$)
3:     $P_{\Delta^+_{root_T}} \leftarrow root_T$.EnumAdded(NIL)
4:     $P_{\Delta^-_{root_T}} \leftarrow root_T$.EnumRemoved(NIL)
5:     $O_T$ += $P_{\Delta^+_{root_T}}$
6:     $O_T$ -= $P_{\Delta^-_{root_T}}$
7:     $u^+$ += $P_{\Delta^+_{root_T}}$
8:     $u^-$ += $P_{\Delta^-_{root_T}}$
```

Then for each T-reduct we run T.UpdateTreeAndOutput(u) on each T, in such an order that if for T-reducts T_1 and T_2, $T_1 \neq T2$ we have that $O_{T_1} = \rho_l$ for some leaf $l \in T_2$ implies that $T_1 < T_2$, meaning that call T_1.UpdateTreeAndOutput(u) is made before call T_2.UpdateTreeAndOutput(u).

Obviously by this ordering, we may not have cyclic tree pairs (T_1, T_2), such that $O_{T_1} = \rho_{l_2}$ for some leaf $l_2 \in T_2$ and $O_{T_2} = \rho_{l_1}$ for some leaf $l_1 \in T_1$.

5.3 Node types and update procedures

5.3.1 Frontier delta enumeration

First we consider the delta enumeration procedures for a node in the frontier. This is not a specific node type, but rather it applies to some specific node types, if and only if these are in F of T. When discussing the specific node types we tell if the frontier enumeration procedure applies. Consider algorithm 7, for some node n and n's parent p in T-reduct T. Observe that since $var(P_n)$ = $var(n)$ we have that $P_n = \rho_n$.

First we look at function n.EnumAdded(t_p). By the definition of output set $P_{\Delta^+_n} \ltimes t_p$, we have that such a call should yield each tuple $t', t' \in P_n \ltimes t_p, t' \notin P'_n \ltimes t_p$.

Now consider the condition of the if clause in line 4. The algorithm iterates over each tuple $t \in \Delta^+_n \ltimes t_p$ and checks whether t joins with a tuple in ρ_n, since if this is the true, then $t \in P_n \ltimes t_p$. Afterwards the function computes whether $t \in P'_n \ltimes t_p$ by taking the count value of t_n and subtracting $\Delta t.count$, which gives the count value of t'. Then if this value is 0, then we have that $t \notin P'_n \ltimes t_p$ and t is yielded.

Now look at n.EnumRemoved(t_p). Such a call should yield each tuple $t', t' \notin P_n \ltimes t_p, t' \in P'_n \ltimes t_p$.

This function iterates over each tuple $t \in \Delta^-_n \ltimes t_p$. As we should have that $t \notin P_n \ltimes t_p$ and $\rho_n = P_n \ltimes t_p$ the algorithm checks whether t does not join with any tuple in $\rho_n \ltimes t_p$. Then if this is the case, if $\Delta t.count$ is negative we have that the count value of t' is positive, hence $t \in P'_n \ltimes t_p$. So if the algorithm reaches line 13 indeed t is correctly yielded.

Algorithm 7 Enumerate frontier node

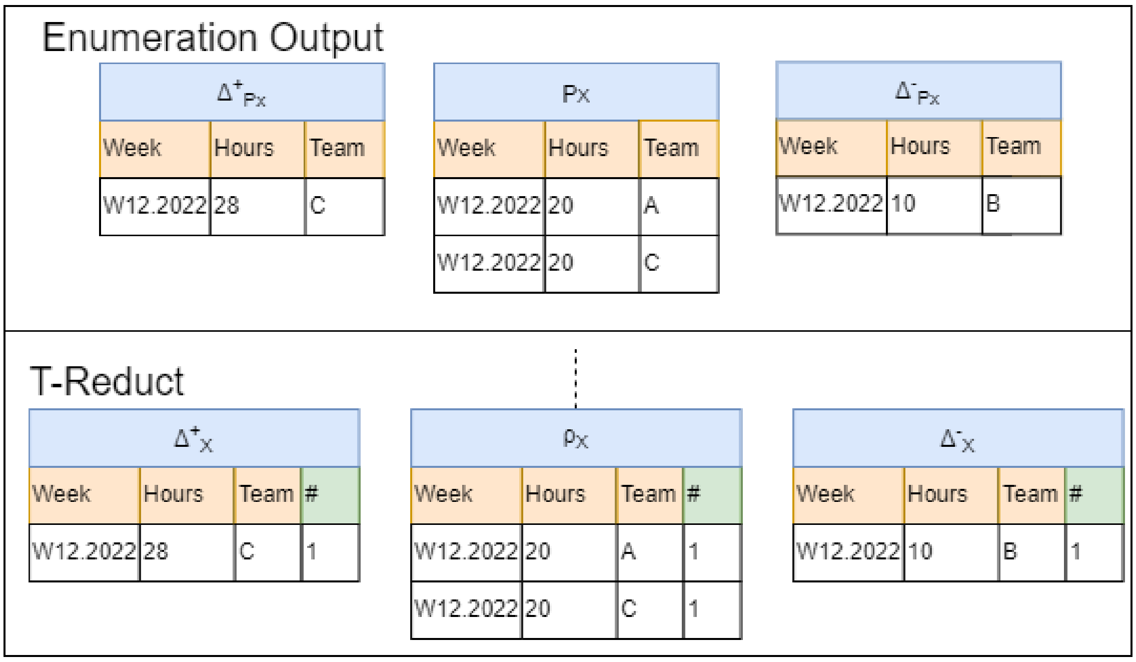
```
 1: function EnumAdded($t_p$)
 2:     for each $t \in \Delta_n^+ \ltimes t_p$ do
 3:         $t_n \leftarrow \rho_n \ltimes t$
 4:         $t_- \leftarrow \Delta_n^- \ltimes t$
 5:         if $t_n \neq NIL \wedge t_n.count - t.count + t_-.count = 0$ then
 6:             yield $t$
 7:
 8: function EnumRemoved($t_p$)
 9:     for each $t \in \Delta_n^- \ltimes t_p$ do
10:         $t_n \leftarrow \rho_n \ltimes t$
11:         $t_+ \leftarrow \Delta_n^+ \ltimes t$
12:         if $t_n = NIL \wedge (t.count - t_+.count > 0)$ then
13:             yield $t$
```

Example

Consider figure 5.1. It shows a leaf node X, which is a child of some node as denoted by the dotted line, together with augmented relations ρ_X, ρ_X^+ and ρ_X^-; and relations P_X, $P_{\Delta_X^+}$ and $P_{\Delta_X^-}$, which are the results of algorithms Enum, EnumAdded and EnumRemoved on node X with tuples NIL respectively.

Observe that, since X is a leaf node, we have that $\rho_X = P_X$, $\rho_X^+ = P_{\Delta_X^+}$ and $\rho_X^- = P_{\Delta_X^-}$, and additionally all tuples in any of these relations have a multiplicity of 1. Then as the enumeration procedure is called on X ánd X is a leaf, we have that $X \in F$.

The Enum algorithm is equal to the enumeration procedure in the original paper, thus simply all tuples in ρ_X are also in P_X. Then as $[W12.2022, 28, C] \notin \rho_X^-$, $[W12.2022, 28, C] \notin \rho_X$ and $[W12.2022, 10, B] \notin \rho_X^+$, we have that at lines 5 and 12 of algorithm 7 the conditions hold and tuples $[W12.2022, 28, C]$ and $[W12.2022, 10, B]$ are in $P_{\Delta_X^+}$ and $P_{\Delta_X^-}$ respectively.



Δ^+_{Px}

Week	Hours	Team
W12.2022	28	C

Px

Week	Hours	Team
W12.2022	20	A
W12.2022	20	C

Δ^-_{Px}

Week	Hours	Team
W12.2022	10	B

Δ^+_X

Week	Hours	Team	#
W12.2022	28	C	1

ρ_X

Week	Hours	Team	#
W12.2022	20	A	1
W12.2022	20	C	1

Δ^-_X

Week	Hours	Team	#
W12.2022	10	B	1

Figure 5.1: A frontier leaf node and its enumeration results.

5.3.2 Join node

The join node corresponds to the type of node from the original paper, in which it is any node n, such that n has two children. In our extension there are also other node types that have two children, however the ComputeDelta and (non-delta) enumeration algorithms from the papers do correspond with those algorithms or cases in the original paper, where n has two children.

Also similarly as in the original paper, a join node may be part of the frontier. In that case not the upcoming algorithms will be the enumeration procedure, but those specified by algorithm 7.

ComputeDelta

The ComputeDelta function found at algorithm 8 is not so much different as the case starting at line 8 in algorithm 1. Rather, it is a bit more specific about the way $\pi_{var(n)}(\rho_m \bowtie \Delta_c)$ is computed.

Now consider algorithm 8. We should still have that $\Delta_n \mathrel{+}= \pi_{var(n)}(\rho_m \bowtie \Delta_c)$. However, since either $var(n) \subseteq var(c)$ or $var(n) \subseteq var(m)$, we have that $\pi_{var(n)}(\rho_m \bowtie \Delta_c) = \pi_{var(n)}(\Delta_c)$ or $\pi_{var(n)}(\rho_m \bowtie \Delta_c) = \pi_{var(n)}(\rho_m \ltimes \Delta_c)$ respectively. Therefore this is exactly what is added to Δ_n at lines 5 and 9 respectively.

Observe that in this algorithm we specified how a *count* value is propagated. In the case of a join node, this is merely a maintenance value. (For the curious reader, when Δ_n is applied - more specifically when a tuple from Δ_n^- is applied and thus this tuple set is removed - the procedure does not necessarily remove the tuple, but rather decrements the *count* value. Then, if this *count* value decrements past 1, the tuple will be removed.) Therefore we may as well not have specified this value in this abstract version of the algorithm. Moreover if not necessary, similar maintenance values will not be specified in algorithm descriptions from here on.

The reason why we chose to specify the value in this case is that it demonstrates the necessity of iterating over all tuples $t2 \in \rho_m \ltimes t1$ at line 4, $t1 \in \Delta_c$. Then what happens is that for each such $t2$, $t1$ is added to Δ_n, instead of checking whether $|\rho_m \ltimes t1| > 0$ and adding $t1$ just one time. This is necessary because the exact count for tuple $\pi_{var(n)}(t1)$ needs to be maintained, which can be only achieved by passing the multiplicity of $t1.count$ and $t2.count$ for each $t2$.

Algorithm 8 Join node ComputeDelta

```
1: function ComputeDelta($c$)
2:     if $var(n) \subseteq var(c)$ then
3:         for each $t1 \in \Delta_c$ do
4:             for each $t2 \in \rho_m \ltimes t1$ do
5:                 $\Delta_n += \pi_{var(n)}(t1), t1.count * t2.count$
6:     else
7:         for each $t1 \in \Delta_c$ do
8:             for each $t2 \in \rho_m \ltimes t1$ do
9:                 $\Delta_n += \pi_{var(n)}(t2), t1.count * t2.count$
```

As the regular Enum function does not differ from the case starting at line 11 of algorithm 2 (apart from the general alterations specified in subsection 5.2.1), we will not consider this algorithm explicitly.

EnumAdded

Consider algorithm 9. We should have that a call n.EnumAdded(t_p) returns $\Delta^+_{P_n} \ltimes t_p$.

Hence it should return each tuple t':
$t' \in P_n \ltimes t_p \wedge t' \notin P'_n \ltimes t_p = t' \in (P_{c1} \bowtie P_{c2}) \ltimes t_p \wedge t' \notin (P'_{c1} \bowtie P'_{c2}) \ltimes t_p$

Then at line 7 the algorithm yields tuples $s1 \cdot s2$, (where $\cdot$ is the concatenation symbol):

$$\begin{array}{lcl} s1 \in c1.\text{EnumAdded}(t) & & \\ \Rightarrow s1 \in \Delta^{+}_{P_{c1}} \ltimes t & \text{and} & \begin{array}{l} s2 \in c2.\text{Enum}(t) \\ \Rightarrow s2 \in P_{c2} \ltimes t \end{array} \\ \Rightarrow s1 \in P_{c1} \ltimes t \wedge s1 \notin P'_{c1} \ltimes t & & \end{array}$$

for all $t \in \Delta^{+}_{n} \ltimes t_p$.

Then as we have that both $s1$ and $s2$ is compatible with some t and moreover either $t \subseteq s1 \vee t \subseteq s2$, we have that $s1$ is compatible with $s2$, which implies that $s1 \cdot s2 \in (P_{c1} \bowtie P_{c2}) \ltimes t_p$. Also $s1 \notin P'_{c1} \ltimes t \Rightarrow s1 \cdot s2 \notin (P'_{c1} \bowtie P'_{c2}) \ltimes t_p$.

Similarly, at line 11 the algorithm yields tuples $s1 \cdot s2$, such that $s1 \in P_{c1} \ltimes t$, $s2 \in P_{c2} \ltimes t \wedge s2 \notin P'_{c2} \ltimes t$, for all $t \in \Delta^{+}_{n} \ltimes t_p$. This also implies that $s1 \cdot s2 \notin (P'_{c1} \bowtie P'_{c2}) \ltimes t_p$.

A tuple t in a join only is added if part of t is added in either child, and we know that if that is the case that by the ComputeDelta procedure $\pi_{var(n)} t$ is added to Δ^{+}_{n}. For this reason it should be intuitive that indeed the EnumAdded function adds all tuples to $\Delta^{+}_{P_n}$ that should be added.

Finally observe that the tuples $s1 \cdot s2$ at line 6 and 10 may have an intersection, for $s1$ and $s2$ such that $s1 \in \Delta^{+}_{P_{c1}} \ltimes t$ and $s2 \in \Delta^{+}_{P_{c2}} \ltimes t$. Since the algorithm should yield unique tuples we check via a hash based method whether $s1 \cdot s2$ has already been yielded.

Algorithm 9 Join node enumerate added tuples

```
1:  function EnumAdded($t_p$)
2:      for each $t \in \Delta^{+}_{n} \ltimes t_p$ do
3:          $alreadySeen \leftarrow$ new HashSet
4:          for each $s1 \in c1$.EnumAdded($t$) do
5:              for each $s2 \in c2$.Enum($t$) do
6:                  $alreadySeen$.Add($s1 \cdot s2$)
7:                  yield $s1 \cdot s2$
8:          for each $s2 \in c2$.EnumAdded($t$) do
9:              for each $s1 \in c1$.Enum($t$) do
10:                 if $s1 \cdot s2 \notin alreadySeen$ then
11:                     yield $s1 \cdot s2$
12:
```

EnumRemoved

Consider algorithm 10. We should have that a call n.EnumAdded(t_p) returns $\Delta^{-}_{P_n} \ltimes t_p$.

Hence it should return each tuple t':
$t' \notin P_n \ltimes t_p \wedge t' \in P'_n \ltimes t_p = t' \notin (P_{c1} \bowtie P_{c2}) \ltimes t_p \wedge t' \in (P'_{c1} \bowtie P'_{c2}) \ltimes t_p$

Then at line 3 the algorithm assigns to MJP tuples $s1 \cdot s2$:
$s1 \notin P_{c1} \ltimes t$, $s1 \in P'_{c1} \ltimes t$, $s2 \in P_{c2} \ltimes t$, $s2 \notin P'_{c2} \ltimes t$. And at line 4 the algorithm assigns to PJM tuples $s1 \cdot s2$:
$s1 \in P_{c1} \ltimes t$, $s1 \notin P'_{c1} \ltimes t$, $s2 \notin P_{c2} \ltimes t$, $s2 \in P'_{c2} \ltimes t$.

In line 9 the algorithm yields tuples $s1 \cdot s2$:
$s1 \notin P_{c1} \ltimes t$, $s1 \in P'_{c2} \ltimes t$, $s2 \notin P_{c2} \ltimes t$, $s2 \in P'_{c2} \ltimes t$. Thus corresponding to when both parts of a tuple are removed.

Then at line 13 the algorithm yields tuples $s1 \cdot s2$:
$s1 \notin P_{c1} \ltimes t$, $s1 \in P'_{c1} \ltimes t$, $s2 \in P_{c2} \ltimes t$ and $s1 \cdot s2 \notin MJP$. From that we may conclude $s2 \in P'_{c2} \ltimes t$. Thus corresponding to the case if part of a tuple is removed from P_{c1}.

Similarly line 17 yields tuples $s1 \cdot s2$:
$s1 \in P_{c1} \ltimes t$, $s2 \notin P_{c2} \ltimes t$, $s2 \in P'_{c2} \ltimes t$ and $s1 \cdot s2 \notin PJM$, (or was already yielded). From that we may conclude $s1 \in P'_{c2} \ltimes t$. Thus corresponding to the case if part of a tuple is removed from P_{c2}.

Thus in lines 9, 13 and 17 the algorithm yields tuples for which it holds that:$s1 \cdot s2 \notin (P_{c1} \bowtie P_{c2}) \ltimes t_p$ and $s1 \cdot s2 \in (P'_{c1} \bowtie P'_{c2}) \ltimes t_p$, as should be the case.

Algorithm 10 Join node enumerate removed tuples

```
function EnumRemoved($t_p$)
    for each $t \in \Delta_n^- \ltimes t_p$ do
        $MJP \leftarrow$ MinusJoinPlus($t$)
        $PJM \leftarrow$ PlusJoinMinus($t$)
        $S1 \leftarrow c1.EnumRemoved(t)$
        $S2 \leftarrow c2.EnumRemoved(t)$
        for each $s1 \in S1$ do
            for each $s2 \in S2$ do
                yield $s1 \cdot s2$
        for each $s1 \in c1.EnumRemoved(t)$ do
            for each $s2 \in c2.Enum(t)$ do
                if $s1 \cdot s2 \notin MJP$ then
                    yield $s1 \cdot s2$
        for each $s2 \in c2.EnumRemoved(t)$ do
            for each $s1 \in c1.Enum(t)$ do
                if $s1 \cdot s2 \notin PJM$ then
                    yield $s1 \cdot s2$

function MinusJoinPlus($t$)
    $S1 \leftarrow c1.EnumRemoved(t)$
    $S2 \leftarrow c2.EnumAdded(t)$
    for each $s1 \in S1$ do
        for each $s2 \in S2$ do
            yield $s1 \cdot s2$

function PlusJoinMinus($t$)
    $S1 \leftarrow c1.EnumAdded(t)$
    $S2 \leftarrow c2.EnumRemoved(t)$
    for each $s1 \in S1$ do
        for each $s2 \in S2$ do
            yield $s1 \cdot s2$
```

Example

Consider figure 5.2. It shows join node A and children B and C. Both B and C are in F, where B is a leaf and C is a join node. Though, we have that since $C \in F$, the enumeration algorithms associated with C are the frontier enumeration algorithms. For each node n the figure shows Δ_n^+ and Δ_n^-, if this is not an empty relation. Finally it shows P_A, $P_{\Delta_A^+}$ and $P_{\Delta_A^-}$.

Note, in our implementation the output relation also contains dimension $C.Week$, which will be removed only after enumerating the root node of the T-reduct, but for simplicity reasons we will omit this dimension for the output relations in this example.

From regular join enumeration algorithm 2 it should be clear why this indeed returns P_A. Now we look how EnumAdded and EnumRemoved return $P_{\Delta_A^+}$ and $P_{\Delta_A^-}$ respectively. We consider three tuples and see why they are, or are not, returned and verify this.

First, consider tuple $['Rik', W12.2022]$. As $['Rik', W12.2022] \in \rho_{\Delta_B^+}$ and $['Rik', W12.2022] \in \rho_2$, both with multiplicity 1, we know that $['Rik', W12.2022] \notin P'_B$ where P'_B is the previous version of P_B. Therefore we know that there is no tuple $t1_{plus} \in P_A$, such that $['Rik', W12.2022] \subset t1_{plus}$, and therefore we know that $['Rik', W12.2022, 20] \notin P'_A$. As $['Rik', W12.2022]$ is compatible with tuple $[W12.2022, 20]$, $[W12.2022, 20] \in P_C$, we have that indeed $['Rik', W12.2022, 20] \in P_A$. Concluding, as $['Rik', W12.2022, 20] \in P_A$ and $['Rik', W12.2022, 20] \notin P'_A$, indeed $['Rik', W12.2022, 20]$ should be in $P_{\Delta_A^+}$, as the figure claims it does.

Also we verify that this is the case. EnumAdded iterates over $[W12.2022]$ as $[W12.2022] \in \Delta_A^+$. Since $['Rik', W12.2022] \in \rho_2$ and $['Rik', W12.2022] \notin \rho'_2$, it holds that $['Rik', W12.2022] \in P_{\Delta_B^+}$ and therefore is returned by B.EnumerateAdded($[W12.2022]$). Then as $t3 = [W12.2022, 20] \in \rho_3$, $t3$ is returned by C.Enum($[W12.2022]$) and at line 7 $['Rik', W12.2022, 20]$ is indeed returned by A.EnumAdded(NIL).

For $['Rik', W12.2022, 20]$ finally observe how it also would be returned at line 11, if it wasn't for the check at line 10. We indeed do not want it to be returned at line 11, as we want each tuple yielded by the algorithm to be unique.

Secondly consider tuple $['Anne', W12.2022, 20] \in P_A$ and observe that it is not in $P_{\Delta_A^+}$. This should indeed be correct because of the following argument.

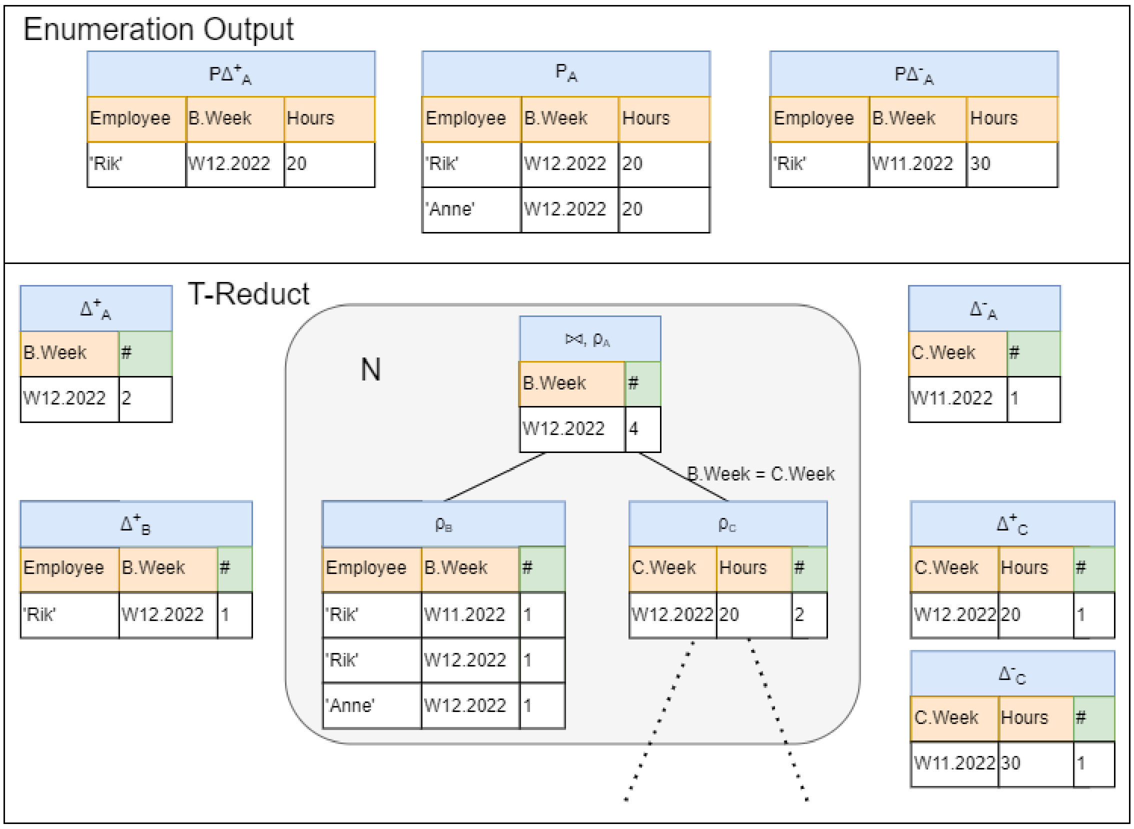
We have that tuple $[W12.2022, 20]$, $[W12.2022, 20] \in \rho_3$ with multiplicity 2 and $[W12.2022, 20] \in \Delta_C^+$ with multiplicity 1, impliying that $[W12.2022, 20] \in \rho'_3$ and therefore $[W12.2022, 20] \notin P_{\Delta_C^+}$. This means that, as $['Anne', W12.2022] \in \rho'_2$, we have that $['Anne', W12.2022, 20] \in P'_A$ and thus should not be added again.

Additionally we verify that $['Anne', W12.2022, 20]$ is indeed not added via A.EnumAdded(NIL). First, $['Anne', W12.2022] \notin P_{\Delta_B^+}$ as it was not added to ρ_2 in this update and therefore is not iterated over in line 4 of algorithm 9. Secondly we established that $[W12.2022, 20] \notin P_{\Delta_C^+}$, meaning that it will not be iterated over in line 8.

Thirdly consider tuple $['Rik', W11.2022, 30] \in P_{\Delta_A^-}$. This tuple is indeed correctly removed.

Since $[W11.2022, 30] \in \Delta_C^-$ and $[W11.2022, 30] \notin \rho_3$, we have that $[W11.2022, 30] \in P_{\Delta_C^-}$ and also $[W11.2022, 30] \in \rho'_3$. Then $['Rik', W11.2022] \in \rho_2$, where $['Rik', W11.2022] \notin \Delta_B^+$, thus $['Rik', W11.2022] \in \rho'_2$. Hence we have that $['Rik', W11.2022, 30] \in P'_A$, and as $['Rik', W11.2022, 30] \notin P_A$, indeed it is removed.

Finally let us verify that in algorithm 10 $['Rik', W11.2022, 30]$ is removed. As $\Delta_B^+ \ltimes [W11.2022, 30] = \emptyset$ we have $P_{\Delta_B^+} \ltimes [W11.2022, 30] = \emptyset$, thus $['Rik', W11.2022, 30] \notin PJM$. We established that $[W11.2022, 30] \in P_{\Delta_C^-}$, thus it is iterated over on line 14. As $['Rik', W11.2022] \in \rho_2$, $['Rik', W11.2022] \in P_B$, thus it is iterated over on line 15. Then, as we saw that $['Rik', W11.2022, 30] \notin PJM$, this tuple is indeed yielded at line 17.



$P\Delta^+_A$

Employee	B.Week	Hours
'Rik'	W12.2022	20

P_A

Employee	B.Week	Hours
'Rik'	W12.2022	20
'Anne'	W12.2022	20

$P\Delta^-_A$

Employee	B.Week	Hours
'Rik'	W11.2022	30

Δ^+_A

B.Week	#
W12.2022	2

$\bowtie, \rho_A$

B.Week	#
W12.2022	4

Δ^-_A

C.Week	#
W11.2022	1

Δ^+_B

Employee	B.Week	#
'Rik'	W12.2022	1

ρ_B

Employee	B.Week	#
'Rik'	W11.2022	1
'Rik'	W12.2022	1
'Anne'	W12.2022	1

ρ_C

C.Week	Hours	#
W12.2022	20	2

Δ^+_C

C.Week	Hours	#
W12.2022	20	1

Δ^-_C

C.Week	Hours	#
W11.2022	30	1

Figure 5.2: A join node and its enumeration results.

5.3.3 Aggregation node

The aggregation node corresponds with each node n in the paper for which it holds that n has one child c. Again similarly as with the join node, for our extension it does not not always hold that if a node has one child that it is an aggregation node, but the ComputeDelta algorithm corresponds to the case in the paper where a node has one child.

In our case it will hold that each aggregation node is in the frontier, however note that the enumeration procedures will not be those of algorithms 7, but algorithms specific to the aggregation node. The reason each aggregation node is in the frontier is that, by the behaviour of ComputeDelta and the enumeration algorithms which are soon to be discussed, it is not necessary to further call enumeration functions on the children as Δ_n and ρ_n hold all information necessary to compute P_n, $P_{\Delta_n^+}$ and $P_{\Delta_n^-}$.

As the ComputeDelta function is very similar to line 11 of 1, we won't discuss it, other than that we want to mention that in case where the aggregation function is count, for each $t \in \rho_n$ $t.count$ needs to be maintained and in the case where the aggregation function is sum or average, additionally $t.sum$ needs to be maintained.

Enumeration

Algorithm 11 describes the procedure for enumerating over a aggregation node with a sum function. We think it should be intuitive how the algorithm would look if a count or average aggregation function would be used instead.

Algorithm 11 Enumerate aggregation (sum) node

```
function ENUMERATE($t_p$)
    for each $t \in \rho_n \ltimes t_p$ do
        yield $t \cdot t.sum$
```

Delta enumeration

Algorithm 12 describes both the EnumAdded and EnumRemoved functions for an aggregation node. Again specifically it describes the procedures for a sum node, but nodes corresponding to a count or average aggregation should be intuitive from the described algorithms.

Observe that instead of iterating over $\Delta_n^+ \ltimes t_p$ or $\Delta_n^- \ltimes t_p$ both EnumAdded and EnumRemoved iterate over $\Delta_n \ltimes t_p$, where $\Delta_n = \Delta_n^+ \cup \Delta_n^-$. The reason for this is that for a tuple t, $t \in \Delta_n^-$ or $t \in \Delta_n^+$ does not indicate that t does not have to be added again or that t does not have to be removed respectively. This opposed to for example the join node where at least for adding tuples to P_n the algorithm only needs to iterate over $P_{Delta_n^+}$, but not over $P_{Delta_n^-}$.

This has as consequence that between updating T and executing the delta enumeration procedures, it is necessary to compute a union of Δ_n^+ and Δ_n^+.

To give some intuition why this is the case, imagine we have relations $R(employee, team, hours)$ and $S(team, hours)$. Now S is an aggregation over R, where R describes the amount of hours an employee works per week, and also describes to which team the employee belongs. Then S describes how many hours per week the employees from each team work in total.

Then when a tuple t is removed from R, this means that for some tuple t', where $t'.team = t.team$ we need to remove t' from S (unless $t.hours$ was 0). This is because t'.sum is not correct anymore. However we also need to add a new tuple for $t'.team$ with the new sum value, at least if the new amount of employees for t' team has not become 0.

EnumAdded

Now we give an intuition why EnumAdded and EnumRemoved are correct.

First consider algorithm 12, function EnumAdded. We should have that a call n.EnumAdded(t_p) returns $\Delta_{P_n}^+ \ltimes t_p$.

Hence it should return each tuple t':
$t' \in P_n \ltimes t_p \land t' \in P_n' \ltimes t_p \Rightarrow t' \in P_{\Delta_n^-} \ltimes t_p$.

Thus it should add tuples that are in the current output set of n and were not in the previous output set of n, unless they were also deleted.

If the condition in line 4 is true, it follows that there is some $t_{node} \in \rho_n \ltimes t_{delta}$, for which there is a compatible tuple t_{output}, $t_{output} \in P_n$.

Now we should show that if $t'_{output} \in P'_n \ltimes t_p$, for some $t'_{output} = t_{output} \Rightarrow t'_{output} \in P_{\Delta_n^-} \ltimes t_p$. In this case it follows that as $t_{output} = t'_{output}$, we have $t_{node} = t'_{node}$ and t_{delta}.count and t_{delta}.sum are both 0.

Now we consider EnumRemoved. We know that at line 8 t_{delta} will be iterated over, as EnumRemoved just like EnumAdded iterates over $\Delta_n \ltimes t_p$. Then t_{node} will be found in line 9. Afterwards the sum and count values corresponding to t'_{node} are computed, which are thus the same as those of t_{node}. Then this tuple with these values together with the count and/or sum values form t'_{output}, which is thus t_{output}, as the delta values are 0. This will be done in lines 13 and 14, as we know that $t_{node} \neq NIL$. Then as $t_{output} \in P'_n$ we have that the count value cannot be 0 and t_{output} will be indeed yielded with the same sum and count values as t'_{output}.

EnumRemoved

Now we show that a call n.EnumRemoved(t_p) returns $\Delta^-_{P_n} \ltimes t_p$. Consider algorithm 12, function EnumRemoved.

Hence it should return each tuple t':
$t' \in P'_n \ltimes t_p \wedge t' \in P_n \ltimes t_p \Rightarrow t' \in P_{\Delta^+_n} \ltimes t_p$.

Hence it should return each tuple that was in the previous output set and is not in the current output set, unless it was also added.

The algorithm iterates over each tuple $t_{delta} \in \Delta_n \ltimes t_p$. Then it finds a node t_{node} compatible with t_{delta} if it is still in ρ_n, or sets t_{node} to NIL otherwise. Then using the delta count and sum, together with the current count and sum value from t_{node} if it still exists, it computes the count and sum of t', $t = t'$. Then if $t' \in P'_n$ we reach line 19 and t_{node} together with the previous aggregation values(s), forming t'_{output}, will be yielded.

Now we have to show that $t_{output} \in P_n \ltimes t_p$ for some $t_{output} = t'_{output} \Rightarrow t_{output} \in P_{\Delta^+_n} \ltimes t_p$. Again as $t_{output} \in P_n$ and $t_{output} \in P_n$ there is some $t_{node} \in \rho_n$ and some $t'_{node} \in \rho'_n$, such that $t_{node} = t'_{node}$.

As line 2 iterates over the same set as EnumRemoved, t_{node} will be found in line 3. Then in line 5, t_{node} and its sum value together form t_{output} and will be yielded. Then since $t_{output} = t'_{output}$, t'_{output} is indeed yielded.

Algorithm 12 Delta enumerate aggregation (sum) node

```
1:  function EnumAdded($t_p$)
2:      for each $t \in \Delta_n \ltimes t_p$ do
3:          $t_{node} \leftarrow \rho_n \ltimes t$
4:          if $t_{node} \neq NIL$ then
5:              yield $t_{node} \cdot t_{node}.sum$
6:
7:  function EnumRemoved($t_p$)
8:      for each $t \in \Delta_n \ltimes t_p$ do
9:          $t_{node} \leftarrow \rho_n \ltimes t$
10:         Declare $count$
11:         Declare $sum$
12:         if $t_{node} \neq NIL$ then
13:             $count \leftarrow t_{node}.count - t.count$
14:             $sum \leftarrow t_{node}.sum - t.sum$
15:         else
16:             $count \leftarrow -t.count$
17:             $sum \leftarrow -t.sum$
18:         if $count \neq 0$ then
19:             yield $t_{node} \cdot sum$
```

Example

Consider figure 5.3. It shows sum node A and leaf node B, augmented by ρ and Δ relations, where $\Delta_A = \Delta^+_A \cup \Delta^-_A$. Also it shows enumeration results P_A, $P_{\Delta^+_A}$ and $P_{\Delta^-_A}$. Observe that $\rho_A \in F$, as for us is always the case for aggregation nodes.

First we make some observations.

Consider relation P_A and how this indeed is retrieved using algorithm 11. Additionally, from Δ^+_B and Δ^-_B it should be intuitive how, using update algorithm 1, Δ_A is constructed. Then, as each tuple in P_A is also in $P_{\Delta_A^+}$, each such tuple was added via EnumAdded. Therefore P'_A is simply the set of all removed tuples, thus $P'_A = P_{\Delta_A^-}$.

Secondly, let's see what tuples should be added and removed to and from P_A. We have that all tuples in ρ_B are grouped by dimension $Team$, where for each $Team$ then the sum of the hours is computed, resulting in an output relation with a tuple for each $Team$ and the number of total $Hours$.

For team $Dev1$ some tuples are removed and added to ρ_B, thus for this team a tuple should be removed and another tuple should be added.

For team $Mar1$ we have that a tuple is removed from ρ_B. As there is still a tuple left in ρ_B, and therefore a number of hours associated with this team, we have that a tuple should be removed, but also a tuple should be added.

For team $Con1$ a tuple was added and not removed, meaning that P'_A did not contain a tuple for this team. Therefore only a tuple should be added.

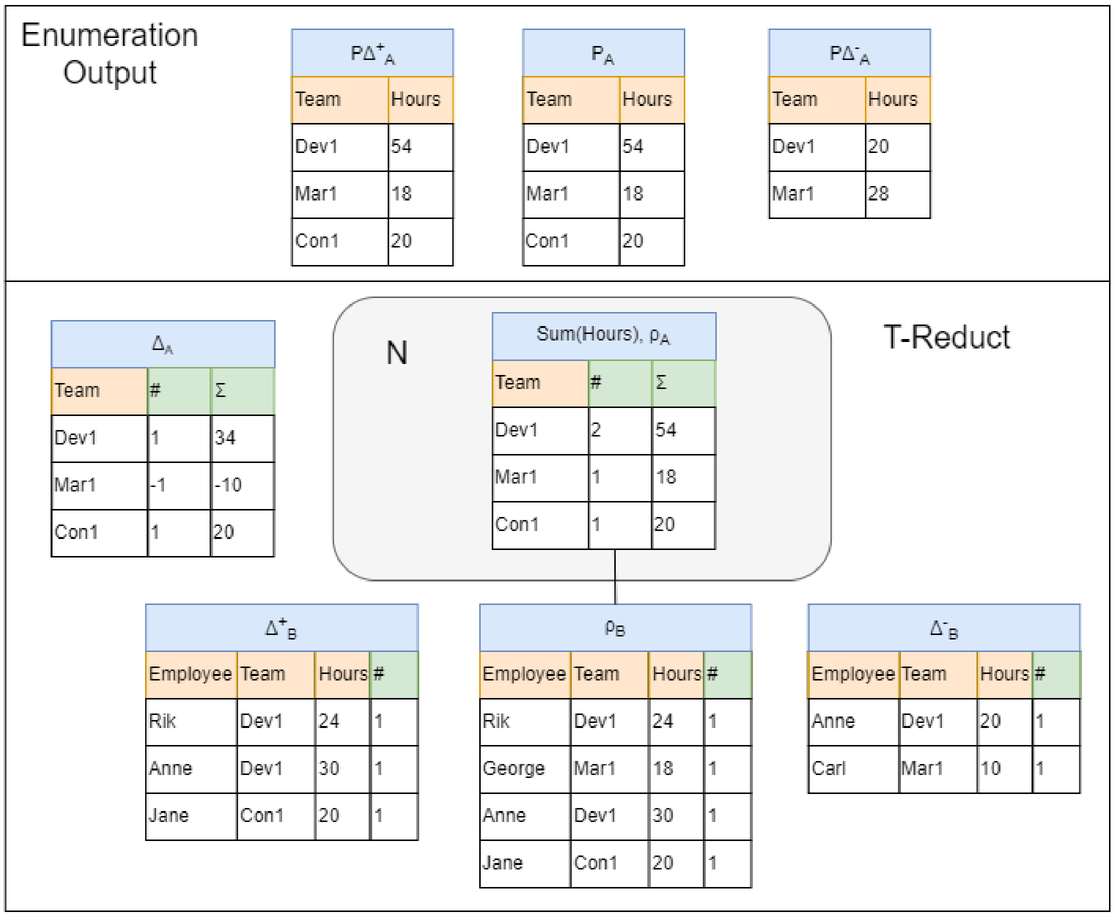
Finally we consider the added and removed tuples.

The tuples in $P_{\Delta_A^+}$ are added via EnumAdded, as shown by lines 1-5 of algorithm 12. The algorithm goes over all tuples in Δ_A and checks whether each such tuple is compatible with a tuple in ρ_A. In the case it is, the algorithm yields a tuple with the $Team$ and Σ value. Observe that for each team $\in \{Dev1, Mar1, Con1\}$ a tuple is returned, as for each team there is a tuple in ρ_A.

For teams $Dev1$ and $Con1$, as there is a compatible tuple in Δ_B^+, it is logical that a tuple is added. For team $Mar1$ however, we see that a tuple is added despite Δ_B^+ not containing a tuple for this team. As for this team a tuple should be added regardless, this example shows why EnumAdded for an aggregation node iterates over Δ_A instead of only Δ_A^+.

The tuples in $P_{\Delta_A^-}$ were removed from P'_A via EnumRemoved, as shown by lines 7-19 of algorithm 12. The algorithm goes over all tuples in Δ_A and checks whether each such tuple was compatible with a tuple in P'_A, where it computes which tuples were in this relation.

For teams $Dev1$ and $Mar1$ the algorithm finds that there used to be compatible tuples in P'_A. Namely, it computes that there was a tuple for $Dev1$ with $Hours$ = 54 - 34 = 20 and that there was a tuple for $Mar1$ with $Hours$ = 18 - -10 = 28. For team $Con1$ however, the algorithm finds that there did not use to be a corresponding tuple in P'_A, as it computes that the *count* value for this tuple would be 1 - 1 = 0, where a ρ relation may not contain such tuples. Thus correctly, for this team no tuple is removed.



$P\Delta^+_A$

Team	Hours
Dev1	54
Mar1	18
Con1	20

P_A

Team	Hours
Dev1	54
Mar1	18
Con1	20

$P\Delta^-_A$

Team	Hours
Dev1	20
Mar1	28

Δ_A

Team	#	Σ
Dev1	1	34
Mar1	-1	-10
Con1	1	20

Sum(Hours), ρ_A

Team	#	Σ
Dev1	2	54
Mar1	1	18
Con1	1	20

Δ^+_B

Employee	Team	Hours	#
Rik	Dev1	24	1
Anne	Dev1	30	1
Jane	Con1	20	1

ρ_B

Employee	Team	Hours	#
Rik	Dev1	24	1
George	Mar1	18	1
Anne	Dev1	30	1
Jane	Con1	20	1

Δ^-_B

Employee	Team	Hours	#
Anne	Dev1	20	1
Carl	Mar1	10	1

Figure 5.3: A sum node and its enumeration results.

5.3.4 Join-Aggregation node

Now we discuss a new type of node, which is a combination of an aggregation and a join node. Hence there is a query in the following general form:
$\pi_{\overline{x},F(y)}(A \bowtie_\theta B)$, for relation $A(\overline{a})$, $\overline{x} \subseteq \overline{a}$, relation $B(\overline{b})$, $y \in \overline{b}$ and some predicate θ.

Necessity of the join-aggregation node

One might wonder why we cannot use a T-reduct composed of a join node together with an aggregation node. Therefore let's reason about this situation.

Assume we have some aggregation node an and join node jn, where an is the parent of jn and jn has children $c1$ and $c2$. Then we have that for each tuple $t1 \in \rho_{c1}$ we have to find corresponding tuples in $t2 \in \rho_{c2}$ and maintain a sum and/or count value for all found tuples $t2$. However the join node update algorithm does not necessarily propagate these aggregation values as it only will propagate the multiplicities of $t1$ and $t2$. This means that jn does not know the aggregation values and cannot propagate these to an, which does need them in order to return P_{an}.

Concluding: joining tuples and propagating the aggregation values needs to be done in one step, thus by one node.

An alternative is that two T-reductions are used, one which joins tuples from $c1$ and $c2$ and a second one that will then aggregation values. This is possible, however it does need the materialization of an extra output relation.

Update and Enumeration

The fact that this type of node is a combination of an aggregation node and join node is also visible when looking at its update and enumeration procedures.

First we look at the update procedure, which is shown at algorithm 13. We should have that, when looking at the general form of a join-over-aggregation query Q, that T corresponding to Q has root p and children $c1$ and $c2$, corresponding to relations A and B respectively, such that $var(p) \subseteq var(c1)$.

Consider algorithm 13. Here the ComputeDelta procedure as we can see is very similar to algorithm 8, only with the extension that the aggregated values are added to the parent tuple.

The latter contains in case the aggregation function is average or sum, the value that needs to be aggregated over, which is part of tuple $t2'$ where $t2' \in \rho_{c2}$ or $t2' \in \Delta^{-}_{c2}$.

In case where the aggregation function is count or average it contains a count value: $t2'.count$. Note that this count value is different from count value $t1'.count * t2'.count$, for $t1' \in \rho_{c1}$ or $t1' \in \Delta^{-}_{c1}$. Also the use of these values are different as $t1'.count * t2'.count$ is used for maintenance, whereas $t2'.count$ is part of the aggregated value.

Algorithm 13 Aggregation over Join node

```
1: function COMPUTEDELTA($c$)
2:     if $var(n) \subseteq var(c)$ then
3:         for each $t1 \in \Delta_c$ do
4:             for each $t2 \in \rho_m \ltimes t1$ do
5:                 $\Delta_n += \pi_{var(n)}(t1), t1.count * t2.count, [t2.aggregation\_values]$
6:     else
7:         for each $t1 \in \Delta_c$ do
8:             for each $t2 \in \rho_m \ltimes t1$ do
9:                 $\Delta_n += \pi_{var(n)}(t2), t1.count * t2.count, [t1.aggregation\_values]$
```

Secondly we consider the enumeration algorithm. Whereas ComputeDelta is (in general) the ComputeDelta for the join node, the enumeration functions are those of the aggregation node, which we showed in algorithms 11 and 12.

As these algorithms are identical to the ones of the Join-aggregation node, there is no reason to discuss them again. However, since the enumeration procedure is that of the aggregation node, we still would like to explicitly notify the reader that therefore the enumeration functions of the join-aggregation node, opposed to those of the join node (that are not in the frontier), do not make enumeration function calls on children. Thus also each join-aggregation node is in F.

Example

Figure 5.4 shows a join-aggregation node A with children B and C. For each node the delta relations are shown, if these are not empty and the result of enumerating the tree is shown.

First we consider algorithm 13 and how it constructs Δ_A. As we denoted earlier, ComputeDelta for a join-aggregation node is very much like that of a regular join node. The difference is that the former precisely denotes that the tuple variables that, once a tuple $t_B \in \rho_B$ or $t_B \in \Delta_B$ is joined with a tuple $t_C \in \rho_C$ or $t_C \in \Delta_C$, that the tuple that is added to Δ_A takes the non-aggregated tuple values from t_B and the aggregated values from t_C.

Then we consider what tuples should be added and removed to and from P_A.

First observe that a tuple with team value $Con1$ and week value 13 is added to ρ_B. However, as this tuple is not compatible with any tuple in ρ_C, no tuple with these values is added to Δ_A.

From ρ_B a tuple with team value $Dev1$ and week value 12 is removed. As it was compatible with tuples in ρ'_c a tuple needs to be removed with this team and week value.

Thirdly we have that a tuple is added to ρ_C with team value $Dev1$ and week value 11. This is compatible with two tuples in ρ_B. The first tuple is t_1 with week value 11. $\rho_c' \ltimes t_1 = \emptyset$, thus for t_1 only a value needs to be added to P_A with the same team and week value. The second tuple is t_2 with week value 12. As t_2 is removed from ρ_A only a tuple needs to be removed from P_A.

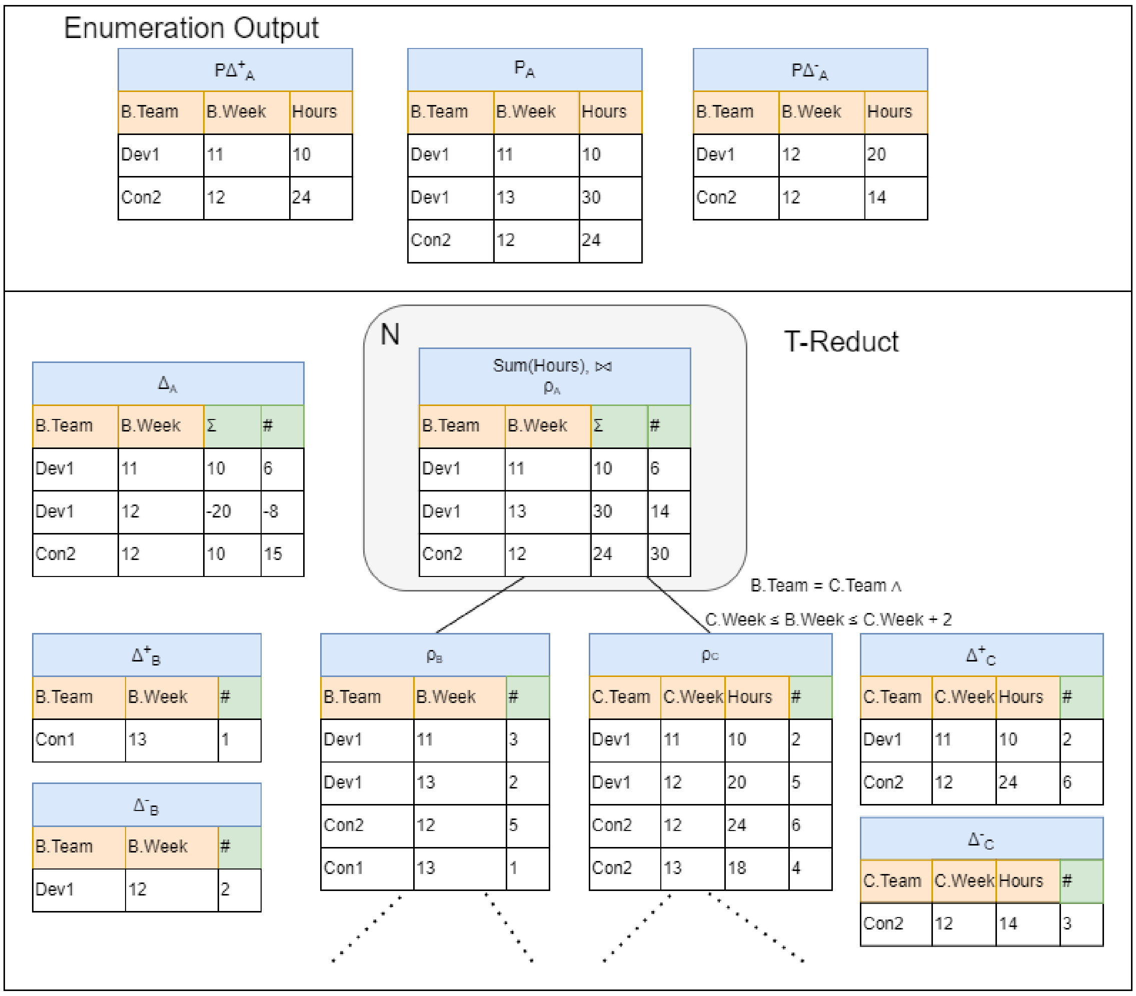
Finally a tuple with team value $Con2$ and week value 12 was updated in ρ_C. As the only compatible tuple in ρ_B has also values $Con2$ and 12, the tuple in P_A with these values needs to be updated as well.

Now we verify that via the regular aggregation node enumeration procedures, as seen in algorithm 12, indeed the correct tuples are added and removed.

First it should be intuitive how P_A is constructed from ρ_A. For the delta enumeration procedures consider the tuples in Δ_A.

Tuples $[Dev1, 11]$ and $[Con2, 12]$ have compatible tuples in ρ_A, thus EnumAdded will return tuples with these tuple values, together with the Σ value as $Hours$ value.

For tuples $[Dev1, 12]$ and $[Con2, 12]$ we have that the *count* value in EnumRemoved does not equal 0. Therefore EnumRemoved will return tuples with these tuple values, together with the Σ value in ρ_A (if there is a compatible tuple in ρ_A), minus the Σ value in Δ_A.



$P\Delta^+_A$

B.Team	B.Week	Hours
Dev1	11	10
Con2	12	24

P_A

B.Team	B.Week	Hours
Dev1	11	10
Dev1	13	30
Con2	12	24

$P\Delta^-_A$

B.Team	B.Week	Hours
Dev1	12	20
Con2	12	14

Δ_A

B.Team	B.Week	Σ	#
Dev1	11	10	6
Dev1	12	-20	-8
Con2	12	10	15

Sum(Hours), ⋈ ρ_A

B.Team	B.Week	Σ	#
Dev1	11	10	6
Dev1	13	30	14
Con2	12	24	30

Δ^+_B

B.Team	B.Week	#
Con1	13	1

Δ^-_B

B.Team	B.Week	#
Dev1	12	2

ρ_B

B.Team	B.Week	#
Dev1	11	3
Dev1	13	2
Con2	12	5
Con1	13	1

ρ_C

C.Team	C.Week	Hours	#
Dev1	11	10	2
Dev1	12	20	5
Con2	12	24	6
Con2	13	18	4

Δ^+_C

C.Team	C.Week	Hours	#
Dev1	11	10	2
Con2	12	24	6

Δ^-_C

C.Team	C.Week	Hours	#
Con2	12	14	3

Figure 5.4: A join-sum node and its enumeration results.

5.3.5 Selection node

A selection node n always has one child c, for which we have that $var(n) = var(c)$, $\rho_n \subseteq \rho_c$ and $P_n \subseteq P_c$. More specifically, n is such that $\rho_n = \sigma_\theta(c)$.

This node is used when a query contains a subquery $\sigma_\theta(R)$, where θ is a selection predicate and R is an atomic relation, such that $P_c = R$. Note that when R is not an atomic relation but a join of multiple relations the subquery will be simplified as described in section 4.8.1.

For this reason we were able to adapt the selection node such that it never is necessary to enumerate c, or its descendants. Therefore the ComputeDelta function, as we soon will see, is adapted such that the algorithm only needs to enumerate n, if $n \in N$. Thus we have $n \in N \Rightarrow n \in F$.

Moreover, as we have that c corresponds to a atomic relation, c is always a leaf, thus its tuples have a multiplicity # of 1. Therefore the enumeration function does not differ from the simple frontier enumeration algorithm described by lines 6 and 7 of algorithm 2. To be precise the regular and delta enumeration algorithm for the selection node are given by algorithm 15.

After considering the ComputeDelta function as described by algorithm 14, together with algorithms 7 it should be intuitive that indeed $\pi_\theta(P_c) \ltimes t_p$, $\pi_\theta(P_{\Delta_c^+}) \ltimes t_p$ and $\pi_\theta(P_{\Delta_c^-}) \ltimes t_p$ are returned by n.Enum(t_p), n.EnumAdded(t_p) and n.EnumRemoved(t_p) respectively.

Algorithm 14 Update selection node

```
function ComputeDelta($c$)
    $\theta \leftarrow$ predicate of edge $(c, n)$
    for each $t \in \Delta_c$ do
        if $\theta$ holds for $t$ then
            $\Delta_n += \pi_{var(n)}(t)$
```

Algorithm 15 Enumerate selection node

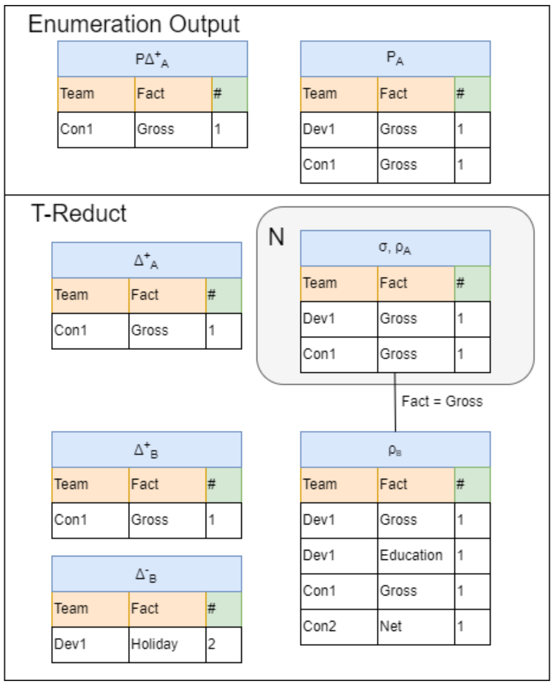
```
function Enum($t_p$)
    for each $t \in \rho_n$ do
        yield $t$

function EnumAdded($t_p$)
    for each $t \in \Delta_n^+$ do
        yield $t$

function EnumRemoved($t_p$)
    for each $t \in \Delta_n^-$ do
        yield $t$
```

Example

Consider figure 5.5. It shows a selection node, together with ρ, delta and enumeration output relations.

The fact that a selection node is always in F, and moreover that B is a leaf, which is the case for the child of any selection node, makes that Enum, EnumAdded and EnumRemoved only have to iterate over ρ_n, Δ_n^+ and Δ_n^- respectively.

These functions do not even have to consider the selection predicate as the ComputeDelta function will only add tuples to the selection node for which the predicate holds. In the example we therefore see that, although relation Δ_B^- is not empty, Δ_A^- ís empty, as for the tuple in Δ_B^- the selection predicate does not hold and therefore did not affect ρ_A' nor affects ρ_A.



$\rho\Delta^+_A$

Team	Fact	#
Con1	Gross	1

ρ_A

Team	Fact	#
Dev1	Gross	1
Con1	Gross	1

Δ^+_A

Team	Fact	#
Con1	Gross	1

σ, ρ_A

Team	Fact	#
Dev1	Gross	1
Con1	Gross	1

Δ^+_B

Team	Fact	#
Con1	Gross	1

ρ_B

Team	Fact	#
Dev1	Gross	1
Dev1	Education	1
Con1	Gross	1
Con2	Net	1

Δ^-_B

Team	Fact	#
Dev1	Holiday	2

Figure 5.5: A selection node and its enumeration results.

5.3.6 Union node

A union node $n \in T$ always has children $c1$ and $c2$, such that $var(P_n) = var(P_{c1}) = var(P_{c2})$ (where the variables are either exactly the same or compared in a predicate with the equal operator). Also n is such that $\rho_n = \rho_{c1} \cup \rho_{c2}$, which also implies that $var(n) = var(c1) = var(c2)$. These properties limit the number of types that $c1$ or $c2$ may be. Namely, for any child $c \in \{c1, c2\}$, we have that $= var(n) = var(c) = var(P_n) = var(P_c)$.

If this would not be the case then there would be a variable $x \in var(P_n), x \notin var(c)$, (as $var(n) \subseteq var(P_n)$). But then as $x \in var(P_c)$, $x \in var(d_c)$, where d_c is some descendant of c. Thus we would have that $x \in var(n) \wedge x \notin var(c) \wedge x \in var(d_c)$, which contradicts the rule that if a variable q in $var(n1)$ and $var(n2)$, for nodes $n1$ and $n2$, then for each node n_i on path $(n1, n2)$ we have that $x \in var(n_i)$.

Because $var(P_n) = var(n)$, $n \in F$ (if $n \in N$) will always hold and the enumeration functions for the union node will always be the frontier enumeration functions as described by algorithm 7.

This node type is used when a query contains a subquery $R1 \cup R2$ for relations $R1$ and $R2$, such that $P_{c1} = R1$ and $P_{c2} = R2$. We assume that $R1 \cap R2 = \emptyset$, as this is the case for the submodels in AC for which corresponding queries contain union operators.

Then algorithm 16 describes the ComputeDelta function for the union node. It should be intuitive from this algorithm, together with enumeration algorithm 7, we have that:

n.Enum(t_p) should return $(R1 \cup R2) \ltimes t_p = (P_{c1} \cup P_{c2}) \ltimes t_p$, which it does.

n.EnumAdded(t_p) should return:

$$\begin{aligned}&((R1 \cup R2) - (R1' \cup R2')) \ltimes t_p \\ &= ((R1 - R1') \cup (R2 - R2')) \ltimes t_p && \triangleright\, R1 \cap R2 = \emptyset \wedge R1' \cap R2' = \emptyset \\ &= (P_{\Delta^+_{c1}} \cup P_{\Delta^+_{c2}}) \ltimes t_p,\end{aligned}$$

and it does.

Thirdly, similar to the previous argument we have that n.EnumRemoved(t_p) should return $(P_{\Delta^-_{c1}} \cup P_{\Delta^-_{c2}}) \ltimes t_p$, and it does.

Algorithm 16 ComputeDelta for a union node

```
1: function ComputeDelta($c$)
2:     for each $t \in \Delta_c$ do
3:         $\Delta_n += t$
```

Example

Consider figure 5.6, which shows a union node A and children B and C, with ρ, delta and output relations. Observe that $n \in F$ and moreover, that any relation has the same set of variables, as is always the case for ρ, delta and output relations corresponding to a union node and its children.

Then for any tuple in the delta relations of B and C, we see that they are simply propagated to Δ_A by algorithm 16 and to P_{Δ_A} by algorithm 7 as we would expect.

5.3.7 Anti-join node

The anti-join node corresponds with a subquery $A \triangleright_\theta B$ for relations A and B, where $t \in A \triangleright_\theta B \Rightarrow t \in A$ and t is not compatible with any tuple in B, thus $t \in A \triangleright_\theta B \implies t \in A \wedge B \ltimes_\theta t = \emptyset$.

However we do not consider $A \triangleright_\theta B$ for any A and B. Namely we only consider A and B such that $var(A) \subseteq var(B)$ and $\triangleright_\theta = \triangleright$, hence it is the natural anti-join. Moreover both A and B have to be atomic relations. These assumptions allow the algorithm to focus on the case where each tuple in B is compatible with one tuple in A, which allows the update and enumeration functions to be a bit more straightforward. Naturally we were able to make this decision for only allowing this subset of anti-join cases, because the submodels that make use of the anti-join operator only need this natural anti-join.

We are convinced that it is possible to extend the following enumeration and update procedures such that they are able to work for any anti-join query, however giving such algorithms is beyond the scope of this project.

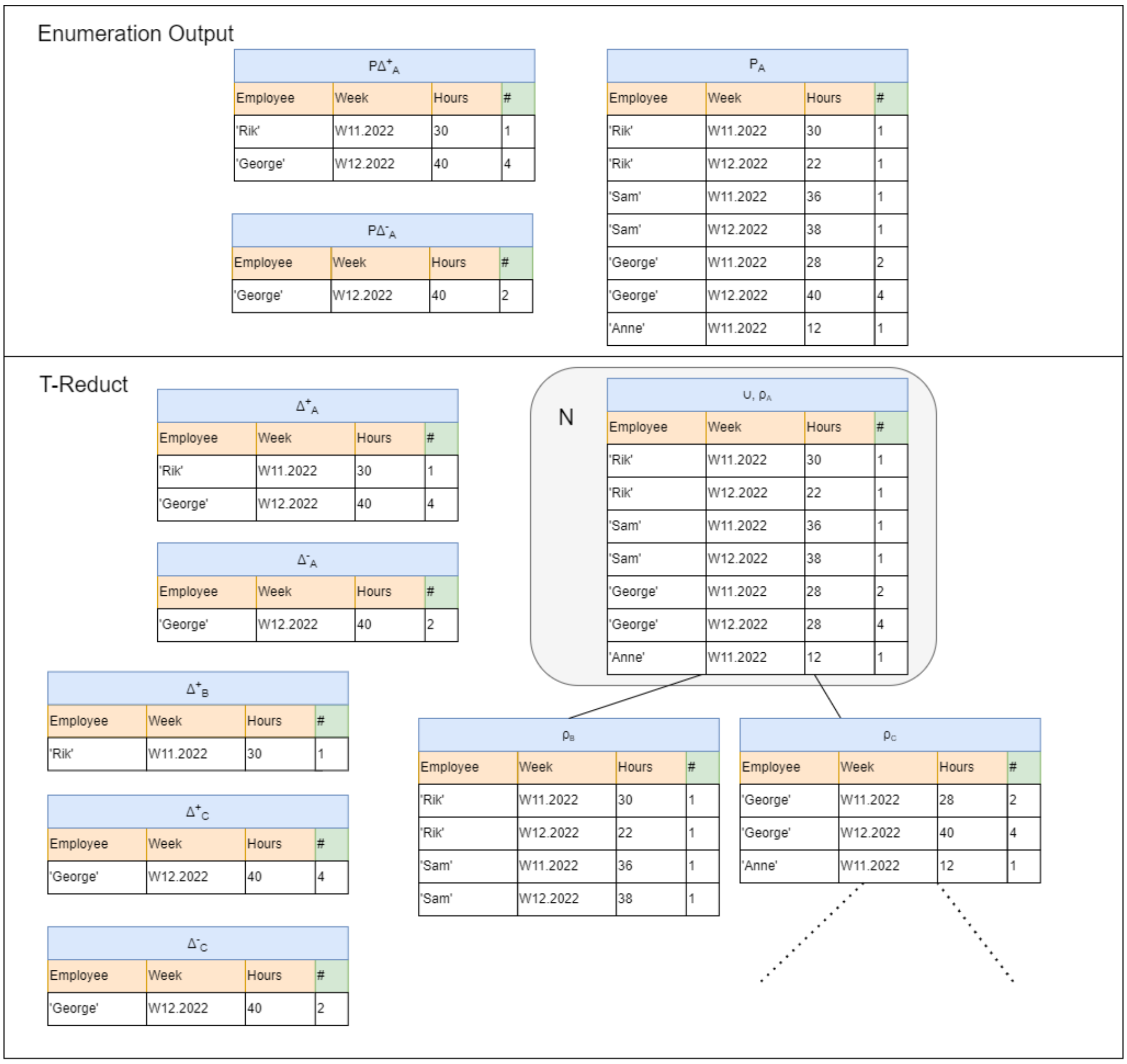
Now we look at some properties of an anti-join node. An anti-join node n has two children $c1$ and $c2$, such that $P_{c1} = A$ and $P_{c2} = B$. Since we consider the natural anti-join we have that $var(n) = var(c1)$ and $var(n) \subseteq var(c2)$. Then, since A is an atomic relation we have that $var(c1) = var(P_{c1})$, which implies that $var(n) = var(P_n)$ and $P_n = \rho_n$. Therefore the enumeration procedure only needs to return ρ_n, thus $n \in F$.

Also, since A and B are atomic relations, we have $\rho_{c1} = P_{c1}$ and $\rho_{c2} = P_{c2}$.

Alternatively, an idea was to design the update and enumeration functions such that $\rho_n = \rho_{c1} \cup \rho_{c2}$. Then the idea was to compute $P_{c1} \triangleright P_{c2}$ on the fly during the enumeration procedures, however this would mean that Enum does not enumerate P_n with constant delay.

The current anti-join enumeration and update procedures differ from the previous procedures in two ways:

1. Instead of three different versions of the enumeration function there are three different versions of the ComputeDelta function. The first function version corresponds to running the update procedure for the first time and is simply called *ComputeDelta*. Then there are two



$P\Delta^{+}_{A}$

Employee	Week	Hours	#
'Rik'	W11.2022	30	1
'George'	W12.2022	40	4

$P\Delta^{-}_{A}$

Employee	Week	Hours	#
'George'	W12.2022	40	2

P_A

Employee	Week	Hours	#
'Rik'	W11.2022	30	1
'Rik'	W12.2022	22	1
'Sam'	W11.2022	36	1
'Sam'	W12.2022	38	1
'George'	W11.2022	28	2
'George'	W12.2022	40	4
'Anne'	W11.2022	12	1

Δ^{+}_{A}

Employee	Week	Hours	#
'Rik'	W11.2022	30	1
'George'	W12.2022	40	4

Δ^{-}_{A}

Employee	Week	Hours	#
'George'	W12.2022	40	2

$\cup, \rho_A$

Employee	Week	Hours	#
'Rik'	W11.2022	30	1
'Rik'	W12.2022	22	1
'Sam'	W11.2022	36	1
'Sam'	W12.2022	38	1
'George'	W11.2022	28	2
'George'	W12.2022	28	4
'Anne'	W11.2022	12	1

Δ^{+}_{B}

Employee	Week	Hours	#
'Rik'	W11.2022	30	1

Δ^{+}_{C}

Employee	Week	Hours	#
'George'	W12.2022	40	4

Δ^{-}_{C}

Employee	Week	Hours	#
'George'	W12.2022	40	2

ρ_B

Employee	Week	Hours	#
'Rik'	W11.2022	30	1
'Rik'	W12.2022	22	1
'Sam'	W11.2022	36	1
'Sam'	W12.2022	38	1

ρ_C

Employee	Week	Hours	#
'George'	W11.2022	28	2
'George'	W12.2022	40	4
'Anne'	W11.2022	12	1

Figure 5.6: A union node and its enumeration results.

functions *ComputeDeltaPlus* and *ComputeDeltaMinus* corresponding to adding tuples to Δ_n^+ and Δ_n^- respectively, for an update u. For the ComputeDelta procedures of other node type, the procedure for adding tuples to Δ_n^+ and Δ_n^- did differ, but not as significantly as ComputeDeltaPlus does differ with ComputeDeltaMinus.

2. There are still 3 different enumeration procedures, however as you can observe in algorithm 17, they differ in the same way that the regular ComputeDelta procedures differ when adding tuples to either Δ_n^+ or Δ_n^-; namely they iterate over a different set of tuples and return a different set of tuples.

Two summarize these 2 alterations, we think it accurate to say that the update and enumeration procedures in some way have switched places. However we think this was necessary, since we did not find an alternative solution, to ensure that the constant delay property of the regular Enum function was upholded.

Another thing to observe is that since EnumAdded and EnumRemoved $(\Delta_n^+ - \Delta_n^-) \ltimes t_{parent}$ and $(\Delta_n^- - \Delta_n^+) \ltimes t_{parent}$, respectively, comparable as the aggregation and join-aggregation node iterates over $(\Delta_n^- \cup \Delta_n^+) \ltimes t_{parent}$ for the enumeration functions.

Then these cases are similar since also for the anti-join node it holds that between the update procedure of T and enumerating over n, Δ_n needs a preprocessing step in which $\Delta_n^+ - \Delta_n^-$ and $\Delta_n^- - \Delta_n^+$ are calculated. This can be done in $\mathcal{O}(|\Delta_n^+| + |\Delta_n^-|)$ time when using a hash-based solution.

It is necessary to iterate over $\Delta_n^+ - \Delta_n^-$ and $\Delta_n^- - \Delta_n^+$, rather than simply Δ_n^+ and Δ_n^- respectively. This is because for one node among $c1$ and $c2$, it holds that it will call ComputeDelta on n, before the other node could apply its own update on itself.

Thus as a consequence say that for example a tuple t may be added to ρ_{c1}, for which a compatible tuple is added to ρ_{c2}. However as $c2$ may not have been updated yet, t then may be added to Δ_n^+ but t should not be in $P_{\Delta_n^+}$. By computing $\Delta_n^+ - \Delta_n^-$ and iterating over this relation in the enumeration procedures, this is ensured.

Algorithm 17 Adding tuples to Δ_{output}^-

```
1: function ENUM($t_p$)
2:     for each $t \in \rho_n \ltimes t_p$ do
3:         yield $t$
4:
5: function ENUMADDED($t_p$)
6:     for each $t \in (\Delta_n^+ - \Delta_n^-) \ltimes t_p$ do
7:         yield $t$
8:
9: function ENUMREMOVED($t_p$)
10:    for each $t \in (\Delta_n^- - \Delta_n^+) \ltimes t_p$ do
11:        yield $t$
```

Now we will discuss the ComputeDelta procedures when computing the initial T-reduct, adding tuples to an existing reduct and removing tuple from a reduct. As always an intuition will be given why the algorithm is correct. However as this algorithm we thought to be the most tricky of all, we decided to prove the correctness of this algorithm. This proof can be found in the appendix.

The proof shows both that all tuples that are added via the algorithm should be added, and tuples that should be added are also added, by mainly considering the ComputeDelta functions, but also the enumerate functions. Then also observe that for any algorithm we need to consider the cases where $c1$ is updated before $c2$ and vice versa.

The following algorithm descriptions will sometimes consider multiple of these update-order cases such that the reader may develop an understanding for the consequences of the order of updating $c1$ and $c2$.

ComputeDelta

First we consider algorithm ComputeDelta which does an initial computation of Δ_p^+ and Δ_p^-, hence $\rho_p = \emptyset$ when running this algorithm. its description can be found in algorithm 18.

Considering the Enum function from algorithm 17, which is the function that will be executed to initially materialize P_p. Observe that this function iterates over all $t \in \rho_n$, where $n = p$. ρ_p will contain all tuples in Δ_p^+ - Δ_p^-. Thus we must have that Δ_p^+ - $\Delta_p^- = P_{c1} - P_{c2}$. Now observe that as $\rho_{c1} = P_{c1}$ and $\rho_{c2} = P_{c2}$, we have that all tuples in $t \in (\rho_{c1} - \rho_{c2}) \implies t \in (\Delta_p^+ - \Delta_p^-)$. This implication also holds the other way around, however this sub-proof can be found in the complete proof in the appendix.

As we know, ComputeDelta(n) will be executed for both $n = c1$ and $n = c2$. However which of $c1$ and $c2$ comes first is arbitrary, which means that the algorithm must work for both cases.

If $c1$ is updated before $c2$, what will happen is that all tuples in ρ_{c1} are added to Δ_p^+ and all tuples in ρ_{c2} that are compatible with a tuple in ρ_{c1} are added to Δ_p^-.

If $c2$ is updated before $c1$, what will happen is that first the update will be applied on ρ_{c2}. Then when executing ComputeDelta with $n = c1$ only those tuples that are not compatible with a tuple in ρ_{c2} will be added to Δ_p^+.

Thus indeed in both cases we have that $t \in (\rho_{c1} - \rho_{c2}) \implies t \in (\Delta_p^+ - \Delta_p^-)$.

Algorithm 18 Update anti-join node initially

```
function ComputeDelta($n$)
    if $n = c1$ then
        for each $t1 \in \rho_{c1}$ do
            if $\rho_{c2} \ltimes t1 = \emptyset$ then
                $\Delta_p^+ += t1$
    else
        for each $t2 \in \rho_{c2}$ do
            $t1 \leftarrow \rho_{c1} \ltimes t2$
            if $t1 \neq NIL$ then
                $\Delta_p^- += t1$
```

ComputeDeltaPlus

Secondly, we consider the algorithm for adding tuples to Δ_p^+. This is shown by algorithm 19.

For case $n = c1$, all tuples are added which were added to ρ_{c1} and are not compatible with a tuple in ρ_{c2}.

For case $n = c2$, the algorithm iterates over all tuples that were removed from ρ_{c2} and looks for an compatible tuple $t1$ in ρ_{c1}. $t1$ surely was not in the anti-join before and therefore also not in ρ_p. But as $t2$ was removed, now $t1$ may be in the anti-join and therefore would have to be added to Δ_p^+. Therefore it looks whether $t1$ still has compatible tuples in ρ_{c2} or not.

Algorithm 19 Adding tuples to Δ_p^+, for anti-join node p

```
function ComputeDeltaPlus($n$)
    if $n = c1$ then
        for each $t \in \Delta_{c1}^+$ do
            if $t \notin \Delta_{c1}^- \wedge \rho_{c2} \ltimes t = \emptyset$ then
                $\Delta_p^+ + = t$
    else
        for each $t2 \in \Delta_{c2}^-$ do
            $t1 \leftarrow \rho_{c1} \ltimes t2$
            if $t2 \notin \Delta_{c2}^+ \wedge t1 \neq NIL \wedge \rho_{c2} \ltimes t1 = \emptyset$ then
                $\Delta_p^+ + = t1$
```

ComputeDeltaMinus

Finally, we consider the algorithm for adding tuples to Δ_p^-, as described by algorithm 20.

For case $n = c1$, the function iterates over all tuples $t1$ that are removed from ρ_{c1}. Then, if $t1$ is not compatible with a tuple in ρ_{c2}, in most cases $t1$ was in the anti-join set and therefore in ρ_p and has to be removed and therefore added to Δ_p^-.

This will at least hold if $c1$ was updated before $c2$, because then $c2$ at this moment still reflects whether, before update u was added, $t1$ was compatible with a tuple in ρ_{c2}.

If $c2$ was updated before $c1$, it might be the case that a tuple $t2$ was removed from ρ_{c2}, such that $t1$ was compatible with $t2$. However in that case, by algorithm 19, $t1$ was also added to Δ_p^+, and since EnumRemoved only iterates over $\Delta_p^- - \Delta_p^+$, $t1$ will not be found.

For case $n = c2$. The algorithm goes over the tuples $t2$ that were added to ρ_{c2} and searches for each $t2$ compatible with a tuple $t1$ in ρ_{c1}. Then it computes the number of tuples that $t1$ was compatible with in ρ'_{c2}: $|\rho'_{c2} \ltimes t1|$, by taking the current count $|\rho_{c2} \ltimes t1|$ and subtracting $|\Delta_{c2} \ltimes t1|$. If this was 0, $t1$ was not compatible with any tuple in ρ'_{c2}, thus it probably used to be in the anti-join set, and has to be removed from ρ_p.

If $c1$ was updated before $c2$, $t1$ might not have been in ρ_{c1}, however in that case, since if $t1$ is added to Δ_p^-, $|\rho'_{c2} \ltimes t1| = 0$, thus by algorithm 19, $t1$ was also added to Δ_p^+. Again as EnumRemoved only iterates over $\Delta_p^- - \Delta_p^+$, $t1$ will not be found.

Algorithm 20 Adding tuples to Δ_p^-, for anti-join node p

```
function ComputeDeltaMinus($n$)
    if $n = c1$ then
        for each $t1 \in \Delta_{c1}^-$ do
            if $t1 \notin \Delta_{c1}^+ \wedge \rho_{c2} \ltimes t1 = \emptyset$ then
                $\Delta_p^- += t1$
    else
        for each $t2 \in \Delta_{c2}^+$ do
            $t1 \leftarrow \rho_{c1} \ltimes t2$
            if $t2 \notin \Delta_{c2}^- \wedge t1 \neq NIL$ then
                $count \leftarrow 0$
                for each $t2' \in \rho_{c2} \ltimes t1$ do
                    $count += t2'.count$
                for each $t2' \in \Delta_{c2}^+ \ltimes t1$ do
                    $count -= t2'.count$
                for each $t2' \in \Delta_{c2}^- \ltimes t1$ do
                    $count += t2'.count$
                if $count == 0$ then
                    $\Delta_p^- += t1$
```

Example

Consider figure 5.7. It shows an anti-join node A with children B and C, together with their ρ and delta relations and output relations of A.As should be the case for an anti-join node, $P_A = \rho_A = \rho_B \rhd_{B.Week=C.Week \wedge B.Team=C.Team} \rho_C$.

It should clear how $P_A, P_{\Delta_A^+}$ and $P_{\Delta_A^+}$ are computed by algorithm 17.

Regarding the ComputeDelta functions in algorithms 19 and 20, consider the tuples in the delta relations of A and B.

First, tuple $[Dev1, W11.2022] \in \Delta_B^+$ is not propagated to $\Delta^+ A$ as it is compatible with a tuple in ρ_C.

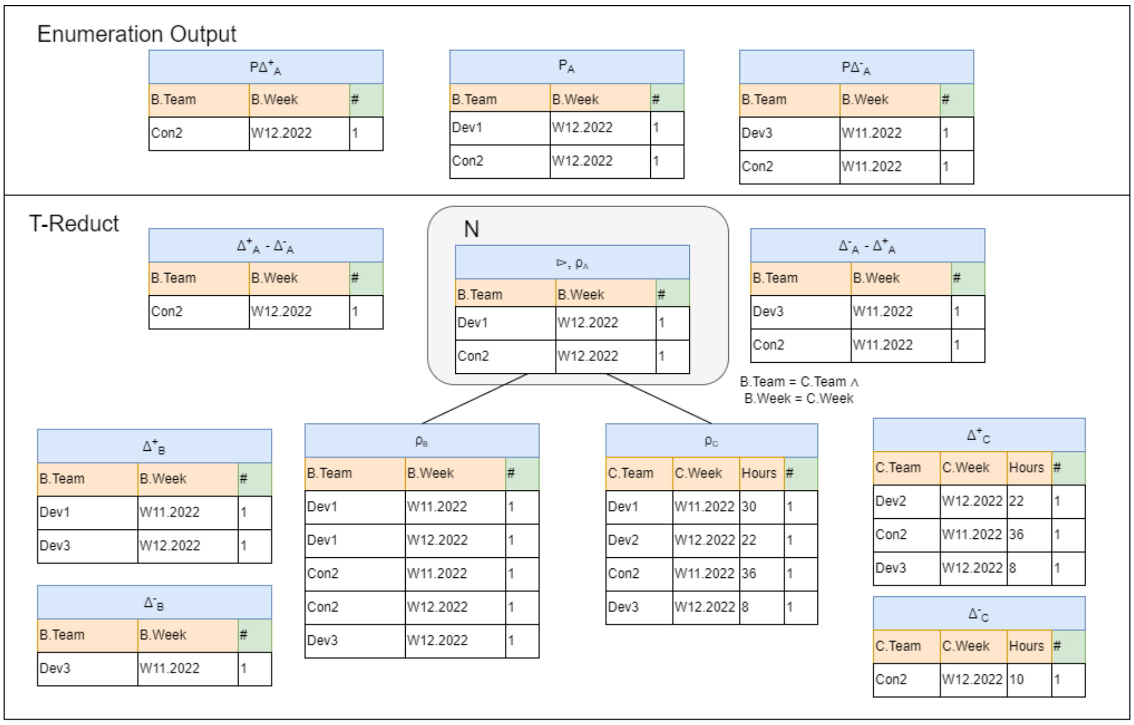
Secondly, tuple $[Dev3, W12.2022] \in \Delta_B^+$ is compatible with tuple $[Dev3, W12.2022, 8] \in \rho_C$. Observe that $[Dev3, W12.2022, 8] \in \Delta_C^+$. Therefore, if ρ_B was updated before ρ_C, $[Dev3, W12.2022]$ was added to Δ_A^+. However if this was the case we have that $[Dev3, W12.2022]$ also was added to Δ_A^-. Otherwise $[Dev3, W12.2022]$ was added to neither Δ_A^+ or Δ_A^-. Observe that regardless, $[Dev3, W12.2022] \notin \Delta_A^+ - \Delta_A^-$ and $[Dev3, W12.2022] \notin \Delta_A^- - \Delta_A^+$. This is what should be the case as $[Dev3, W12.2022]$ should neither be removed from P'_A, as $[Dev3, W12.2022] \notin P'_A$, and $[Dev3, W12.2022]$ should not be in P_A as $[Dev3, W12.2022] \notin \rho_B - \rho_C$.

Thirdly, tuple $[Dev3, W11.2022] \in \Delta_B^-$ is added to Δ_A^- as it was or is not compatible with any tuple in ρ'_C or ρ_C respectively, and therefore used to be in $\rho_B \rhd \rho_C$.

Then, tuple $[Dev2, W12.2022, 22] \in \Delta_C^+$ is or was not compatible with any tuple in ρ_B or ρ'_B respectively, thus no compatible tuple is added to Δ_A^+.

Δ_C^+ also contains tuple $[Con2, W11.2022, 36]$. This tuple is compatible with $[Con2, W11.2022]$, which did not use to be compatible with any tuple in ρ'_C, thus $[Con2, W11.2022]$ is added to Δ_A^-.

Finally tuple $[Con2, W12.2022, 10] \in \Delta_C^-$ is compatible with tuple $[Con2, W12.2022]$, which is not compatible with any tuple in ρ_C. Thus $[Con2, W12.2022]$ is added to Δ_A^+.



$\rho\Delta^{+}_{A}$

B.Team	B.Week	#
Con2	W12.2022	1

ρ_{A}

B.Team	B.Week	#
Dev1	W12.2022	1
Con2	W12.2022	1

$\rho\Delta^{-}_{A}$

B.Team	B.Week	#
Dev3	W11.2022	1
Con2	W11.2022	1

$\Delta^{+}_{A} - \Delta^{-}_{A}$

B.Team	B.Week	#
Con2	W12.2022	1

$\rhd, \rho_{A}$

B.Team	B.Week	#
Dev1	W12.2022	1
Con2	W12.2022	1

$\Delta^{-}_{A} - \Delta^{+}_{A}$

B.Team	B.Week	#
Dev3	W11.2022	1
Con2	W11.2022	1

Δ^{+}_{B}

B.Team	B.Week	#
Dev1	W11.2022	1
Dev3	W12.2022	1

Δ^{-}_{B}

B.Team	B.Week	#
Dev3	W11.2022	1

ρ_{B}

B.Team	B.Week	#
Dev1	W11.2022	1
Dev1	W12.2022	1
Con2	W11.2022	1
Con2	W12.2022	1
Dev3	W12.2022	1

ρ_{C}

C.Team	C.Week	Hours	#
Dev1	W11.2022	30	1
Dev2	W12.2022	22	1
Con2	W11.2022	36	1
Dev3	W12.2022	8	1

Δ^{+}_{C}

C.Team	C.Week	Hours	#
Dev2	W12.2022	22	1
Con2	W11.2022	36	1
Dev3	W12.2022	8	1

Δ^{-}_{C}

C.Team	C.Week	Hours	#
Con2	W12.2022	10	1

Figure 5.7: An anti-join node and its enumeration results.

5.4 Conversion procedure of queries to GJTs

As depicted by figure 5.8, the procedure of converting an Assemble sub-model to one or more T-reducts consists of three sub-procedures. First the sub-model needs to be converted to a query in the relational algebra format, secondly such a query needs to be converted to a GJT and finally the GJT is extended to a T-reduct.

For the first sub-procedure there is no algorithm known yet, however the method that we used for converting the AHA model is described in section 3.3. Finding such an algorithm we consider future work.

The algorithm for the third procedure on the other hand is given by the paper and did not need any alterations from an algorithmic point of view.

Finally, the second sub-procedure, thus converting a query to a T-reduct is rather trivial as the GJT follows the queries operators very closely. To show this we compare an example of a query and it's GJT counterpart.

The query shown is the second query of section 3.3.2, namely using a relation DC001, it computes the net availability of an employee per week, which then will be stored DC001 as well.

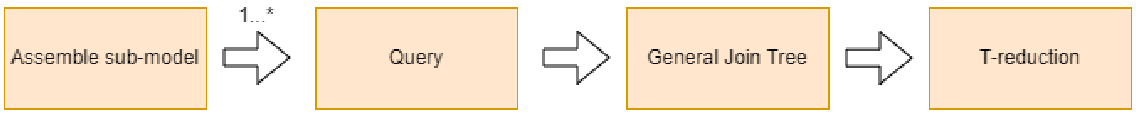


Figure 5.8: Sub-model to T-reduct/T-reductions

The query then is:

$$
\begin{aligned}
& DC001_2 \leftarrow DC001_1 \cup \\
& \pi_{a.employee,d.fact,a.week,hours \leftarrow (a.hours - b.hours - c.hours)} \\
& \quad (\rho_a(\sigma_{fact=grossavail.}(DC001_1)) \bowtie_{a.employee=b.employee \wedge a.week=b.week} \\
& \quad \rho_b(\sigma_{fact=holiday}(DC001_1)) \bowtie_{b.employee=c.employee \wedge b.week=c.week} \\
& \quad \rho_c(\sigma_{education}(DC001_1)) \times \\
& \quad \rho_d(Factnet))
\end{aligned}
$$

The GJT that describes this query is shown in figure 5.9, together with two extra nodes that are not part of the GJT itself, but describe operations that need to be performed to derive the output relation. The first of these operations is the projection step. The second operation is unifying the enumerated result with the instance of DC001 without the *net availability* tuples.

The reason why these nodes are not part of the GJT is that those nodes will not be augmented with relations ρ and Δ. Therefore they are thus also not considered in the update procedure and will not be called upon by the enumerating procedure, at least not until the very end. Here they will follow a fixed procedure as these operations always need to be performed at the end, although sometimes the output relation already is the complete output relation for the model and thus does not need to be unified with another relation.

However note that the GJT may contain union nodes as we saw in section 5.3.6. As we know such a node ís updated and enumerated.

Observe that the properties for the GJT T can be generated from the query Q itself.

- Each $n \in T$ corresponds to a query operator in Q.
- Each leaf node $l \in T$ corresponds to an input relation in Q. var(l) corresponds to the dimensions of the input relation, where the variables have the name of the value name together with an alias if the relation has an alias in Q.
- Each selection node $sn \in T$ and it's child c share an edge showing an predicate that is the same as a selection predicate in Q.
- Each join node $jn \in T$ has children $c1$ and $c2$ and shows a join predicate θ at edge (jn, $c2$) that corresponds to a join predicate in Q. The variables of jn are then $var(c1) \cap var(\theta)$.
- The frontier is decided exactly in the way described by the paper. Namely from the frontier we also know N, which is such that $var(N) = out(Q)$, where $out(Q)$ is the set with the projected variables of Q. However for certain node-operator types we have that they always are in the frontier, if they are part of N. These are the select, aggregate and join-aggregate nodes.

Cyclic queries

It may occur that a sub-model requires multiple T-reductions such that the sub-model's output relation can be computed by DYN. We know this is the case if a query is cyclic, for which AC has one example, namely sub-model 8 which computes the moving total.

This model computes query:

DC001$_a$(fact, team, a.week.w, a.week.y, a.hours) $\bowtie$ DC001$_b$(fact, team, b.week.w, b.week.y, b.hours) $\bowtie$ Yearlengths(b.week.y, b.length),

for which we can describe the join predicates as superset:

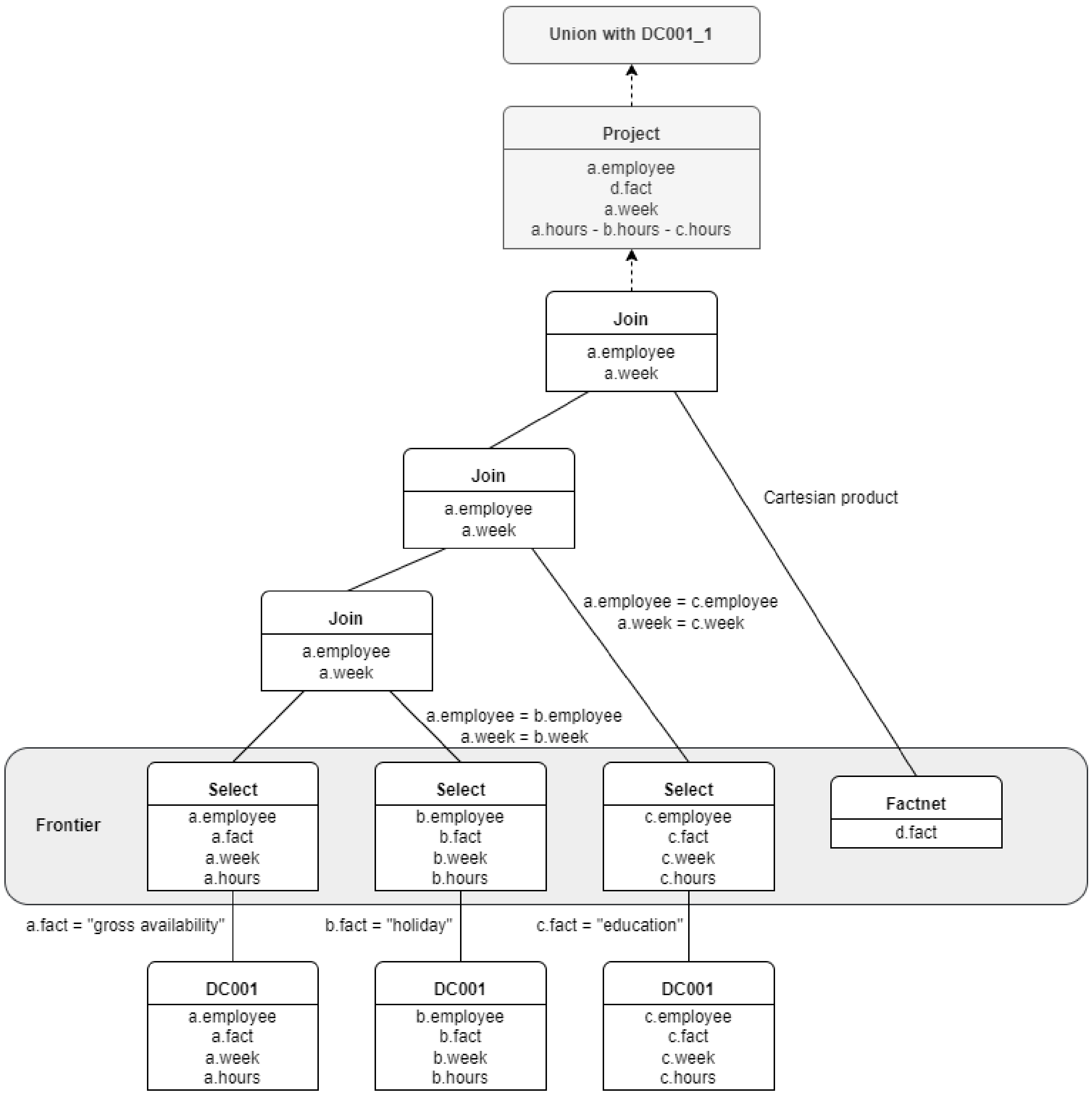


Figure 5.9: GJT for computing the net availability

θ = {{a.fact, b.fact},{a.team, b.team},{a.week.y, b.week.y}, {a.week.w, b.week.w}, {a.week.w, b.length, b.week.w}, {b.week.y, year}}.

θ consists of variable sets, where for each comparison there is a variable set with the variables in that comparison.

Then the predicate with variables {a.week.w, b.length, b.week.w} contains a variable from each input relation, such that we cannot have a GJT such that for each inner node p it has a child c for which $var(p) \subseteq var(c)$ ánd that if a variable v is in two nodes $n1$ and $n2$, then there is a path from $n1$ to $n2$ where each node on this path contains v.

Therefore for this sub-model there are two general join trees, corresponding to two queries, as we see at section 3.3.2, query 8.

5.5 Evaluation

We now will evaluate the introduced algorithms and the implementation of these and the general IVM procedure for AC. First we consider the general time complexity for delta enumeration, we then consider efficiency of applying updates to relations, we look at whether our algorithms and implemention is feasible for the current state of Assemble and finally we draw conclusions, where we answer research sub-questions 2 and 3.

5.5.1 Delta enumeration time complexity

For the regular enumeration procedure we know that we iterate the output set with constant delay. For the delta enumeration procedure this will not be the case as tuples in a delta enumeration function will only conditionally be yielded, meaning that the time between yielded tuples may vary. However we may still have that the running time is linear in the size of Δ_{root_T}. Therefore we will examine the running time of the delta enumration procedure to see whether this is the case.

First we look at the individual n.EnumDelta(t_p) call, for a node $n \in T$, made by the enumeration procedure. Each such call starts with an iteration over $t \in X_n \ltimes t_p$ for some set X_n. Then let k be the biggest $|X_n \ltimes t_p|$ for any combination of X_n and t_p that will be encountered during the delta enumeration procedure, apart from the one called in the root, as we know there that, as t_p is NIL, $|X_n \ltimes t_p| = |X_n|$. Let x be the amount of tuples iterated over for for a call n.EnumDelta(t_p) Let $\delta(n)$ be the depth of a node n, with $\delta(root_T) = 0$ and d be $\max_{n \in N}(\delta(n))$.

Consider the following cases for an arbitrary node n:

1. n is in the frontier. Then we need to consider the aggregate, anti-join and frontier enumeration procedures. Such a call n.EnumDelta(t_p) iterates over $\mathcal{O}(k)$ tuples. For each such tuple there will be an operation in constant time, thus the running time for both EnumAdded and EnumRemoved is $\mathcal{T}[n, t_p] = x$.
2. If n is a union node, the running time for both EnumAdded and EnumRemoved is: $\mathcal{T}[n, t_p] = x * (\mathcal{T}[c1, t_p] + \mathcal{T}[c2, t_p])$.
3. Also, n may be a join node. For EnumAdded the running time is $\mathcal{T}[n, t_p] = x * ((\mathcal{T}[c1, t_p] + |P_{\Delta^+_{c1}} \ltimes t_p| * \mathcal{T}(c2.\text{Enum}(t_p))) + (\mathcal{T}[c2, t_p] + |P_{\Delta^+_{c2}} \ltimes t_p| * \mathcal{T}(c1.\text{Enum}(t_p))))$.
4. Finally n may be a join node. Then for EnumRemoved the running time is $\mathcal{T}[n, t_p] = x * ($
$(\mathcal{T}[c1, t_p] + \mathcal{T}[c2, t_p] + |P_{\Delta^-_{c1}} \ltimes t_p| * |P_{\Delta^+_{c2}} \ltimes t_p|)$
$+ (\mathcal{T}[c1, t_p] + \mathcal{T}[c2, t_p] + |P_{\Delta^+_{c1}} \ltimes t_p| * |P_{\Delta^-_{c2}} \ltimes t_p|)$
$+ (\mathcal{T}[c1, t_p] + \mathcal{T}[c2, t_p] + |P_{\Delta^-_{c1}} \ltimes t_p| * |P_{\Delta^-_{c2}} \ltimes t_p|)$
$+ (\mathcal{T}[c1, t_p] + |P_{\Delta^-_{c1}} \ltimes t_p| * \mathcal{T}(c2.\text{Enum}(t_p)))$

$+ (\mathcal{T}[c2, t_p] + |P_{\Delta^-_{c2}} \ltimes t_p| * \mathcal{T}(c1.\text{Enum}(t_p))))$
$\leq \mathcal{O}(x * 5 * (\mathcal{T}[c1, t_p] + \mathcal{T}[c2, t_p] + |P_{\Delta^{+|-}_{c1}} \ltimes t_p| * |P_{\Delta^{+|-}_{c2}} \ltimes t_p|))$

Observe that for a $n \in F$ the running time is upperbounded by case 1. For a node in $N \setminus F$ this upperbound is case 4. Now we should find out the number of calls made at an arbitrary depth.

In the root we make two calls: $root_T$.EnumAdded(NIL) and $root_T$.EnumRemoved(NIL). Then each such call makes (as upperbound) $\mathcal{O}(10 * |\Delta_{root_T}|)$ calls, where each such call in turn makes $\mathcal{O}(10 * k)$ calls. Thus on depth l, $l > 0$, there are $\mathcal{O}(10 * |\Delta_{root_T}| * (10 * k)^{l-1})$ calls made in total.

Then for a tuple in Δ_{root_T} we make $\mathcal{O}(10 * (10 * k)^{d-1}) \leq \mathcal{O}((10 * k)^d)$ calls, and in general a call by a node on depth l makes $\leq \mathcal{O}((10 * k)^{d-l})$ calls in the leaf nodes. This means that $|P_{\Delta^{+|-}_c} \ltimes t_p| \leq \mathcal{O}(\#$ of leaf calls from $\mathcal{T}[c, t_p]) = \mathcal{O}((10 * k)^{d-l})$. Thus $|P_{\Delta^{+|-}_{c1}} \ltimes t_p| * |P_{\Delta^{+|-}_{c2}} \ltimes t_p|$ $= \mathcal{O}(((10 * k)^{d-l})^2) \leq \mathcal{O}(((10 * k)^d)^2) = \mathcal{O}((10 * k)^{2d})$. Hence it follows that each node in $N \setminus F$ has a cost of $\mathcal{O}((10 * k)^{2d})$.

Thus the running time for a tuple in Δ_{root_T} is:
# of inner call nodes * cost + # of leaf call nodes * cost
$\leq (\sum_{0 \leq l \leq d-1}(\mathcal{O}((10 * k)^l)) * \mathcal{O}((10 * k)^{2d})) + (\mathcal{O}((10 * k)^d) * \mathcal{O}(1))$
$\leq \sum_{0 \leq l \leq d}(\mathcal{O}((10 * k)^l)) * \mathcal{O}((10 * k)^{2d})$
$= \mathcal{O}(\frac{(10*k)^{d+1}-1}{10*k-1}) * \mathcal{O}((10 * k)^{2d})$
$\leq \mathcal{O}((10 * k)^{d+1}) * \mathcal{O}((10 * k)^{2d})$
$= \mathcal{O}((10 * k)^{3d+1})$

Now examine this upperbound for one tuple Δ_{root_T}: $\mathcal{O}((10 * k)^{3d+1})$. We have that d in AC is upperbounded by constant 3. Then if k is also upperbounded by some constant, we should have that the total running time is $|\Delta_{root_T}| * \mathcal{O}((10 * k)^{3d+1}) = |\Delta_{root_T}| * \mathcal{O}(1)^{\mathcal{O}(1)} = |\Delta_{root_T}| * \mathcal{O}(1)$ $= O(|\Delta_{root_T}|)$.

Concluding, assuming k is a constant, we have that the running time $\mathcal{T}[\Delta_{root_T}, NIL]$ is linear in the size of Δ_{root_T}, thus $\mathcal{T}[\Delta_{root_T}, NIL] = \Delta_{root_T} * c$, for some constant c. This will be verified by experiments.

5.5.2 Inefficiency of applying updates

Our implementation suffers a major inefficiency in respect to applying updates to the ρ relations in the T-reductions. This apply-update step can be found in algorithm 3 at line 7. This inefficiency comes from the fact that index $I_{n'}$, for some relation $n' \in \{\rho_n, \Delta^+_n, \Delta^-_n\}$, is designed such that semi-joins for tuples belonging to the parent node or sibling node of n, can be performed in constant, or logarithmic time. This is achieved by having that $var(I_{n'}) = var(p)$, where $var(p) \subseteq var(n)$. Then, both adding and removing a tuple t to some ρ_n, involves searching some tuple $t' \in \rho_n$, $t = t'$. For adding t we then replace t' by a tuple with the same values, but count: t.count + t'.count. Or if t' does not exist we simply add t. For removing t we subtract t.count from t'.count and remove t' if the count is smaller than 1.

Then this search operation must be achieved by using I_{ρ_n}, which thus has variables $var(p)$, such that for each t to be added or removed, we need to iterate through a relation $\rho'_n \subseteq \rho_n$, $t' \in \rho'_n \iff \pi_{var(p)}(t) = \pi_{var(p)}(t')$. Theoretically $|\rho'_n| = O|\rho_n|$. The biggest size occuring in an AC T-reduct is the amount of dimension values for dimensions fact, team and week multiplied = $4 * 12 * 52 = 2496$, which is in the T-reduct for joining datasets R(fact, team, week.w, week.y, hours) with S(week.y, length), and adding and removing tuples to R. This can be found in query 8 at section 3.3.2.

In order to improve the running time we modify each index I_s in T, with key set K, relation S and hash function $f_K()$, $\exists_{t \in S} \Rightarrow \exists_{k \in K} \wedge t \in f_K(k)$, to index I^2_s. I^2_s is a double-layered index, such that I^2_s consists of: key set $K' = K$, hash function $f_{K'}$, $S' = S$, key set $K2$ and hash function $f_{K2}()$:

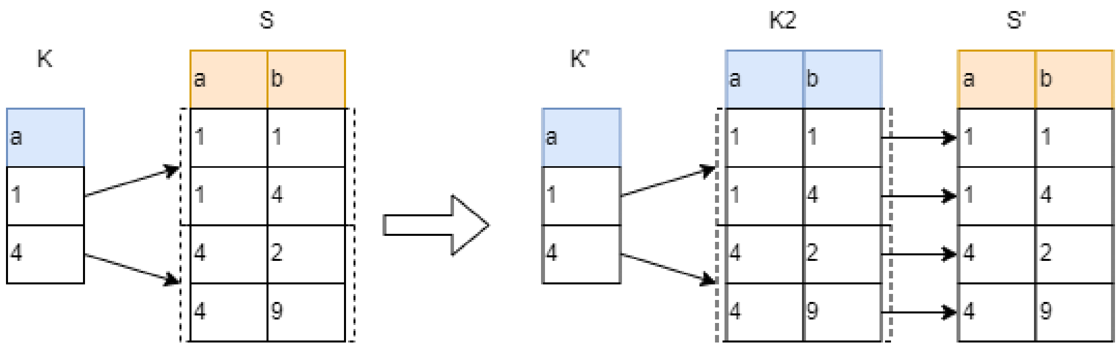


Figure 5.10: Single layered index to double-layered index

$$\exists_{t \in S} \wedge s \in f_K(k), k \in K \iff \exists_{t' \in S', k' \in K, k2 \in K2},\ t = t' \wedge k = k' \wedge k2 \in f_{K'}(k') \wedge s' = f_{K2}(k2)$$

This idea is illustrated by figure 5.10. The downside of this double-layered index is that for each X_n, $X_n \in \{\rho_n, \Delta_n^+, \Delta_n^-\}$ we add an hashset of size $\mathcal{O}(|X_n|)$. However it does allow the algorithm to perform insertions and deletions in $\mathcal{O}(1)$ time.

Search operations do also occur in the enumeration procedure. Namely at the delta enumeration functions of the aggregation node and the frontier node. These may also be affected by the inefficiency of the single-layered index. However, the way we implemented the algorithms for the AHA application, is such that they are not affected. Namely, in our T-reduction, aggregation nodes are always root nodes. For a root it holds that $var(root_T) = var(I_{root_T})$, as $root_T$ does not have a parent. For the frontier enumeration, we did not implement the algorithm shown by algorithm 7, but instead for a call $f.EnumDelta(t_p)$, for $f \in F$ and $EnumDelta \in \{EnumAdded, EnumRemoved\}$ we simply return $\Delta_f^+ \ltimes t_p$ and $\Delta_f^- \ltimes t_p$ respectively.

We did not implement this double-layered index, meaning that we were not able to verify these performance results, however since it theotically seems very possible it should therefore potentially also be in practice. Since then we have that for applying an update Δ, we can do this in $\mathcal{O}(|\Delta|)$ time, we can compute the potential performance of Dynamic Yannakakis for a node n by replacing the time that applying an update Δ_n to ρ_n takes by $|\Delta_n|$.

However since it has not been implemented we should also consider the inefficient running time, where we can compare the implemented 'inefficient' version to the potential running time.

5.6 Storing data in the Oracle database

For now we mainly considered the first two research questions, however the third research question: 'Considering the way Assemble is intertwined with the Oracle OLAP DML, would Dynamic Yannakakis be a practical short-term solution?', may thus for the foreseeable future, be the most important one.

The term 'practical' may be a bit vague, but it is ambiguous on purpose as in our case it has two definitions. The first definition is that Dynamic Yannakakis can be used on demand, without having to have done significant preparations. Thus, can Dynamic Yannakakis be used in an efficient manner to update output datasets, where we have datasets stored in Oracle, an update u and a query representation, thus also a GJT, but not a T-reduct? This would be convenient as all data structures apart from the T-reduct are already being computed or in the case of a GJT, can be computed once as it is independent of the data, and therefore does not count as a 'significant preparation'.

The answer to this question is no, for Dynamic Yannakakis to compute output changes efficiently, T-reductions need to be maintained such that they are up-to-date.

A second definition of 'practical' is that the Oracle database is used for storing the data, as for the foreseeable future this will remain the case. Then what would be the implications? We therefore should define what the data is that Dynamic Yannakakis needs to store.

We have for each T-reduct T, that for each node n relation ρ_n is maintained and reused. Thus these ρ relations need to be stored. Also the output relation for each T needs to be stored, but this was already the case. Finally observe that as for a node n relation Δ_n is made empty for each update, we do not have to store this relation. Additionally this implies that Δ_n relations can be implemented by means of a set of tuples, as it has been up until this point.

Thus a point of great importance is whether datasets in Oracle are suited to be maintained as ρ relations by Dynamic Yannakakis. These will then replace the indexed relation data structures that were used. Also note that since the double-layered indices were only necessary for ρ relations, this idea would not have to be investigated further. We first look how Oracle datasets can be accessed and altered by Assemble. As we have seen earlier, such datasets consists of cells. Such a cell is defined and accessed by a set of dimension values.

Then an important question is whether for a tuple, efficiently semi-join operations can be performed, which we thus use in the ComputeDelta and the enumeration algorithms. Also we have to know whether updates can be quickly applied, but as a tuple that needs to be applied simply contains the dimension values of the cell it targets, this should be no problem.

At first glance this should be no problem, assuming that we do not have only the option to retrieve certain cells for relation R, but multiple cells, where a search tuple t contains values for a subset of all dimensions, thus $var(t) \subset var(R)$. For example, say we want to find all cells in a data cube with dimensions $Employee$, $Fact$ and $Week$, that join with tuple [employee 'Rik', week 'W06.2022']. Then we want to be able to retrieve all cells for Employee 'Rik' and Week 'W06.2022' for all Fact values, without having to know these values. For now it seems that if and only if this is the case, then semi-joins can be computed efficiently using this type of dataset.

Hence, it may very well be the case that these Oracle data cubes are suitable to be used as ρ relations in Dynamic Yannakakis. The fact that such data structures, which look like multi-dimensional arrays, were not used in the paper, is that they only make sense if for almost every combination of dimension values, there is a non-na value. This is very often the case for Assemble datasets, but certainly not in general for each relation, and the paper is aimed at general relations.

There is also a negative implication for using the approach of storing and retrieving data from the Oracle server, but it is also simply inherent to this approach. Namely, having to retrieve (parts of) the ρ relations may be time consuming.

In conclusion, finding out whether these Oracle data cubes are suitable as data structures for relations will need a lot of theoretical, but also practical implications to be considered. Therefore it is beyond the scope of this project and further research needs to be performed, such that a true answer to the third research question can be found. The only definitive answer we can give to this question for now is that, the current version of Dynamic Yannakakis, where relations are always sets of tuples, is not a short-term practical solution.

5.7 Conclusion

We may conclude that Dynamic Yannakakis can be extended such that:

1. A T-reduct can be composed of types of nodes, which correspond to certain query operators. Using such a T-reduct DYN can return the output relation corresponding to the query, by running the update and enumeration procedures corresponding to these nodes.

2. For all node types we also described delta enumeration procedures, (and in the case of the

anti-join node multiple ComputeDelta procedures), such that updates to the output set of a T-reduct can be efficiently returned, where enumeration can be done linearly in the size of the root delta set.

3. There exists a general procedure for propagating updates through a model, where updates to output relations of queries - which are input relations of other queries - can be used to efficiently update these input relations.

Therefore we can answer the question: 'Can we extend Dynamic Yannakakis to facilitate IVM for Assemble models, without nullifying the benefits of the current Dynamic Yannakakis algorithm?', with a resounding yes, in theory this is possible. When doing experiments we will verify whether our implementation, and a version that would apply updates to relations in linear time, indeed is efficient.

Secondly, we may answer the question 'Considering the way Assemble is intertwined with the Oracle OLAP DML, would Dynamic Yannakakis be a practical short-term solution?'. This answer is that the current version, where ρ relations are sets of tuples, is not a practical short-term solution. However if DYN can be adapted, such that it uses the multi-dimensional arrays from the Oracle database, then it may be a short-term practical solution. However finding out whether this is the case we consider future work.

Chapter 6

Experiments

The goal of this chapter is to verify the performance results of the extended version of Dynamic Yannakakis for the Assemble models, which is done by running experiments on our implementation. In this way we can answer the question 'Can we extend Dynamic Yannakakis to facilitate IVM for Assemble models, without nulli-fying the benefits of the current Dynamic Yannakakis algorithm?'. These experiments consist of three smaller experiments. Each experiment should provide an answer to one of the following questions:

1. Is the performance of Dynamic Yannakakis independent from the input relation size?
2. How does Dynamic Yannakakis compare to Assemble in terms of efficiency?
3. Does the delta enumeration procedure running time grow linearly with the root delta size?

This chapter is structured in the following way: First we explain our methodology, such that it is clear what it is that we measure and compare. Secondly we show the results of the experiments, and discuss them, such that we concludingly may give a definitive answer to the second research question.

At this point, it may be smart to recall our most prominent use case; namely, someone who uses an Anago application wants to quickly have output datasets after a small update $|u|$ to an input dataset I is applied. We assume that such an update is quick, if the amount of used cells is small. A small update size we consider to be less than 10 cells, however we will look at update sizes up to 150 cells, such that we also can look at how the number of used cells grows with the update size.

Special attention is paid to when the input relations are of a large size $|I|$, as in that case, Assemble's method of computing the output dataset is not very efficient. This inefficiency stems froms the fact that the amount of used tuples depends on $|I|$ and not on $|u|$. Therefore, before comparing DYN to Assemble and showing that it is more efficient than the current implementation - at least for a large $|I|$ and small $|u|$ - we want to show that for DYN the opposite is true, thus that the running time depends on $|u|$ and not on $|I|$.

6.1 Methodology

We now explain how the experiments where conducted.

6.1.1 Input dataset sizes

For several of the following experiments, we differentiate between input relation sizes. Therefore we should look at what these sizes depends on.

Now recall that the AHA application contains datasets that are datacubes with dimensions such as week, employee, team, fact and value hours. Then when we look at this from a relational point of view, this means that for each combination of the dimensions there is only one hour value. The dimensions week, team and fact are upperbounded by constants $(52 + 1) * 2 = 106$, $8 + 3 + 1 = 12$ and $4 + 2 = 6$ respectively.

Employee is the only dimension that thus is not bounded by some constant. Therefore we may let the input dataset size grow by letting the amount of employees grow. Then, during the experiment we define the input relation size by the amount of employees: n $\in \{10, 100, 1000\}$.

6.1.2 Update definition

We must specify exactly what is an update u, such that we also know what $|u|$ is. We consider u from the Assemble point of view; namely u consists of cell updates. Here a cell update means that some cell c in an Assemble dataset had some old value v', which is changed to another value v. It is not possible to remove or add cells to an Assemble dataset directly. It is possible, however we would have to remove or add a dimension value. This has as consequence that if the dataset is a data cube $dc(a, b, c)$, when adding or removing a dimension value to or from dimension c, we add or remove $a * b$ cells respectively.

For convenience sake, during experiments we only use updates to cells, which correspond to pairs of an addition of a tuple and a deletion of a tuple. However in the T-reduct the relation between an added and deleted tuple corresponding to an updated cell is not known or used by the algorithm, thus conceptually it does not make a difference that we do not use singular added or removed tuples.

6.1.3 Counting

In order to understand how we compare the performance of both Assemble and Dynamic Yannakakis we show how we counted the amount of cells and tuples respectively.

Assemble

For Assemble there is no feasible way to count the amount of used cells during running time. However we have still been able to extract this amount from Assemble. Although we could not make Assemble log the explicit count of tuples, we were able to make Assemble log the selection sizes for each dimension for each dataset that was used in executing a sub-model.

Then each sub-model computes its output dataset by iterating over some datasets, where for each dataset the sub-model iterates over the selected range-combination for the dimensions of the dataset. Thus say that the sub-model iterates over dataset D with dimensions $var(D) = \{a, b, c\}$ with selection sizes x, y and z respectively, then the sub-model for D will iterate over $\#I_D = x * y * z$ cells.

For each sub-model we then needed the knowledge over which datasets it iterates. Lets define this set of datasets with $\mathcal{D}$. We may in this way compute $\#SM = \sum_{D \in \mathcal{D}} \#I_D$, for each sub-model. Then by taking the sum of each $\#SM$ we know the total amount of iterated, and therefore used cells for an Assemble model.

This sum of $\#SM$ values only applied to a run with a specific amount of employees. However, each $\#I_D$ for submodel S, only depends on D's selection sizes for each dimension in $var(D)$, which in our model are independent of the amount of updated cells. Therefore we could easily, without increasing the input datasets sizes and doing additional model executions. We could simply recompute $\#I_D$ for $n \in \{10, 100, 1000\}$, by substituting the amount corresponding to dimension employee with n, if employee$\in var(D)$.

6.1.4 Dynamic Yannakakis

Now we take a look at how for DYN the amount of tuples is counted. In general, the idea is to count each tuple that is accessed. This may be for example in an iteration or in a binary search. We will in two ways count the amount of accessed tuples for DYN.

The first way is by counting all accessed tuples in the current implementation, this current implementation we will call DYN, as we did before.

The second way is to count the number of tuples in the more optimal case, where for a T-reduct, on the ρ relations a double-layered index is implemented and the index on the output relation is altered, such that updates Δ_r on ρ_r can be applied in $\mathcal{O}(|\Delta_{\rho_r}|)$ time, instead of $\mathcal{O}(|\Delta_r| * |\rho_R|)$ time. Then let this theoretical algorithm be called DYN*.

Although we are convinced DYN* is very feasible, since we were not able to implement and test it in time, we still have to consider DYN as it is the only implemented version for which we can verify its correctness and performance.

Now we are going to go over a call T.UpdateTreeAndOutput(u) as described by algorithm 6. It consists of three procedures for which we need to count the amount of accessed tuples.

Apply Update

During the Update procedure - see line 7 of algorithm 3 - and when updating the output dataset in lines 5 and 6 of algorithm 6, updates are applied on ρ_n and O respectively. Adding and removing tuples is implemented by iterating over each $t \in \Delta_n^+$ or $t \in \Delta_n^-$ respectively. Then for each such t we find a list l of tuples by finding a hash bin in I_{ρ_n}, where t serves as a key for the hashtable. As this list is not ordered (unless there is more than one inequalitiy), it iterates one by one through l, until t', for which $t = t'$ has been found. When we add a tuple the count of t' is increased and if no such t' is found t is simply inserted in l. If the algorithm is removing a tuple it will always find such t', will decrease its count and if this count then is 0 t' is removed from l.

In case of DYN, we count each tuple in Δ_n and each tuple in l that is iterated over. For DYN*, we calculate the apply update performance by using the size of Δ_n. Then $2 * |\Delta_n|$ is indeed the number of accessed tuples, as we assume that in a double layered index finding a tuple t' for each $t \in \Delta_n$ accesses two tuples if such t' is found and one tuple otherwise.

ComputeDelta

For ComputeDelta DYN and DYN* are counted in the same way. As we know, the precise n.ComputeDelta(c) function depends on the type of node that n is. However, what each such function shares is that it iterates over $t \in \Delta_c$. Then for each such t the count is incremented by 1. Then for each such t the algorithm may iterate over another set: $S \ltimes t$, for some set S.

First the semijoin time is calculated, however the semijoin is implemented such that for ComputeDelta it is always a constant time (or logarithmic time in case of ¿ 1 inequalties) operation, since S is a relation over the sibling of c. Thus no tuples are counted then. For each tuple that the algorithm iterates over in a semijoin the count is incremented by 1.

Enumeration

For the enumeration procedure the count values for DYN and DYN* are calculated in a similar way. Each call n.EnumF(t_p) iterates over a set each $t \in X_n \ltimes t_p$, $t_p \in \{\rho_n, \Delta_n^+, \Delta_n^-, \Delta_n, \Delta_n^+ - \Delta_n^-, \Delta_n^- - \Delta_n^+\}$. Then for each such t we increment the count by 1. If n is in the frontier, which may hold for an aggregate, join-aggregate and anti-join node, a tuple may be searched for, and if it is found it is also accessed and therefore needs to be counted. The semi-join operation for seaching such a tuple will then always be a constant time operation.

Concluding, we have that DYN is the total number of tuples accesssed during enumeration, the

ComputeDelta function and applying updates. DYN* is the number of tuples accessed during enumeration and the ComputeDelta function, together with the sum of all delta relation sizes.

6.2 Results and discussion

6.2.1 Independency from input relation size

What we want to find out is whether the efficiency of Dynamic Yannakakis is independent from the input relation size. Therefore we look whether the number of used tuples grows more quickly for bigger input relation sizes.

First we look at the performance of our implementation: DYN. Therefore consider figure 6.1. The data that underlies the graph displayed in this figure - and other figures in this chapter - can be found in the appendix. Perhaps when we compare n = 10 to bigger sizes n = 100 and n = 1000, one may have the suggestion that a smaller input size indicates a significantly smaller number of accessed tuples. However when we compare n = 100 to n = 1000, we see that a smaller input size does not indicate less accessed tuples. Hence we may conclude that the efficiency of DYN does not decrease for a larger (input) relation size, and if it does only up to a certain input relation size.

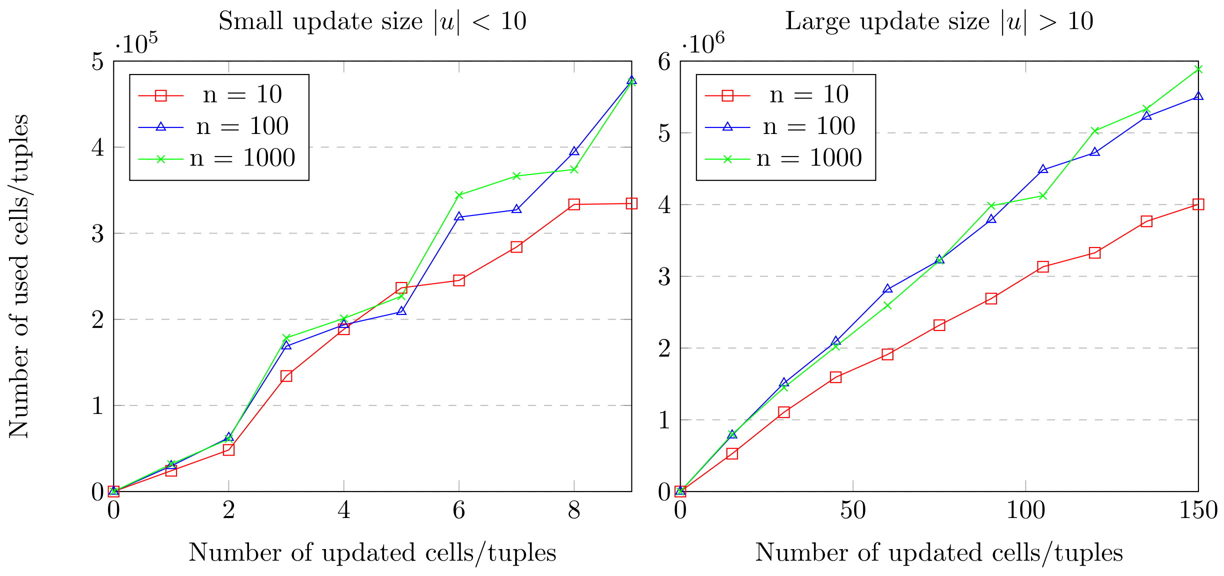


Figure 6.1: Comparing DYN performance per input relation size, for small and big update sizes

Secondly, consider figure 6.2 in which we see the performance of DYN*. Here there is even less doubt that the number of used tuples does not grow with the total (input) relation size; the efficiency for n = 100 compared to n = 1000 per $|u|$ now seems totally arbitrary and also the line for n = 10 is really close to those of n = 100 and n = 1000.

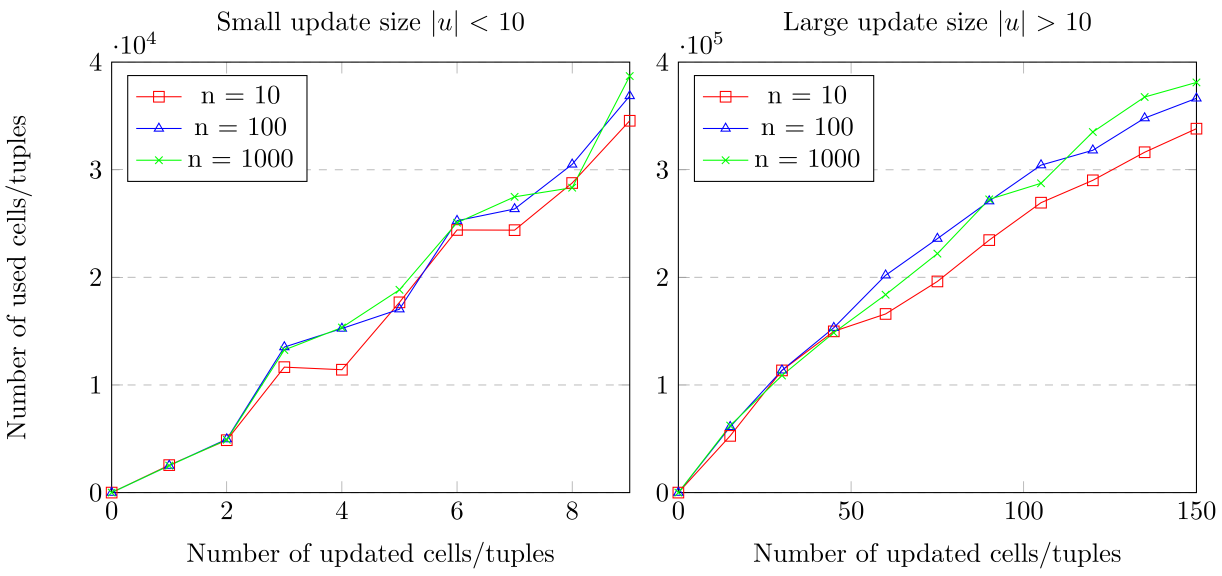


Figure 6.2: Comparing DYN* performance per input relation size, for small and big update sizes

Therefore we may conclude that, unlike Assemble, the order of used tuples for DYN and DYN* does not depend on the input relation size up until some n = 1000. Moreover, since we do not observe any growth from some n upwards, we conclude that the order of used tuples is independent of n in general.

6.2.2 Comparing Dynamic Yannakakis to Assemble

Now we will look at how the efficiency of DYN and DYN* compares to that of Assemble. As the efficiency of Assemble **does** depend on the input relation size, we look again at a small, medium and large input relation size one by one.

First consider figure 6.3. At this point we are able to point out how enormous the difference is between DYN and DYN*. The difference comes only from the potential implementation of more efficient indexing, such that ρ and output relations take much less time to be updated with delta relations. Thus we now also can conclude that for DYN the current running time is dominated by the time for applying updates, as the enumeration and ComputeDelta procedures, are exactly the same for DYN and DYN*.

Next to this we may observe that for a small $|I|$: $n = 10$, we have that the number of accessed tuples for DYN grows quickly towards that of Assemble. For a medium sized and large $|I|$, this number is significantly smaller however. Thus the only case where Dynamic Yannakakis would be beaten by Assemble is for a small $|I|$ and if the performance of DYN* does not turn out to be realistic, however even in that case we have that, since $|I|$ is small and Assemble's performance depends on $|I|$, Assemble's method suffices.

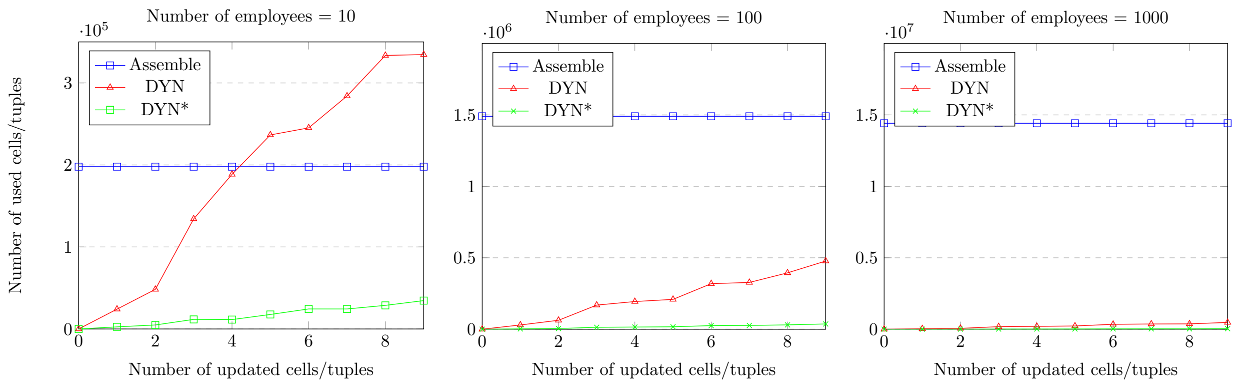


Figure 6.3: Comparing Assemble, DYN and DYN*, for $|u| < 10$

As the performances of DYN and DYN* do not depend on $|I|$, it is not really the question whether less tuples are accessed, but how many compared to Assemble, and how quickly this number grows towards that of Assemble. Therefore, for update sizes $|u| > 10$, we look at the ratio for both DYN and DYN* to Assemble, for small, large and big $|I|$.

Consider figure 6.4. For each increase in the order of $|I|$, we see an increase in the order of difference in efficiency between Dynamic Yannakakis and Assemble.

First, for n = 10, as DYN already was not more efficient than Assemble for a small $|u|$, we will only consider the ratio for DYN*. We see that here around a $|u|$ of 60 that DYN* becomes as efficient as Assemble.

Secondly, for n = 100, we see that DYN surpasses Assemble in the number of used tuples around $|u| = 30$. DYN* however does remain well below the number of used tuples in Assemble.

Finally, for n = 1000, at $|u| = 150$ DYN uses less than half of the number of tuples compared to Assemble. At that point the number of accessed tuples in DYN* does not even come close to a tenth of the accessed cells in Assemble.

These results should not be surprising. Namely, as the number of used tuples for DYN and DYN* barely grows for an increasing n, whereas for Assemble this grows linearly with n, this complies with the factor 10 of difference in ratio between n = 100 and n = 1000, as this factor indeed is 1000 / 100.

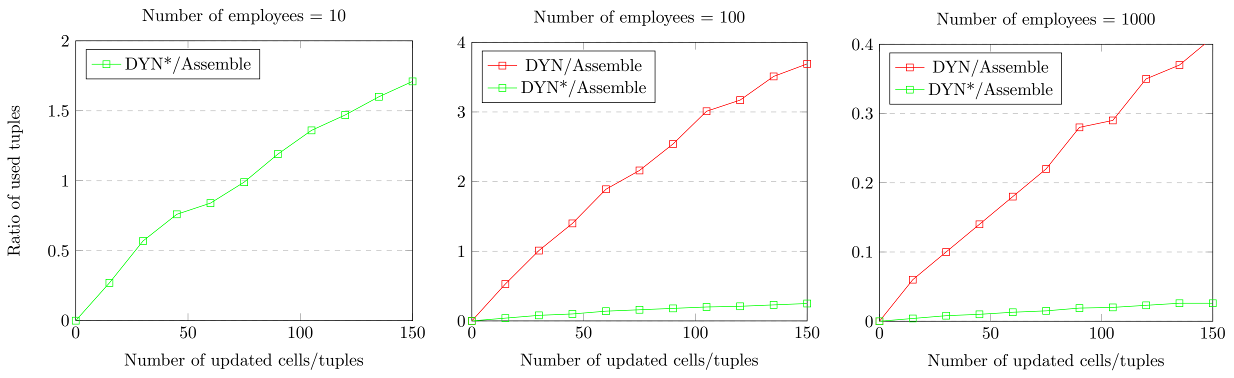


Figure 6.4: Comparing Assemble, DYN and DYN*, for $|u| > 10$

6.2.3 Enumeration

Another thing that we want to verify is whether indeed the running time of the delta enumeration procedure grows linearly with Δ_{root_T}. Therefore for each $|u|$, we take the sum of $|\Delta_{root_T}|$ for all trees T. Let this sum be $\Delta_{\mathcal{R}}$. Then additionally we consider the sum of all counted tuples during the enumeration procedures $\mathcal{E} = \sum \mathcal{T}[\text{T.Enum}]$.

Then if $\mathcal{E}$ grows linearly with $\Delta_{\mathcal{R}}$ we have that $\mathcal{E} = \mathcal{O}(\Delta_{\mathcal{R}})$ and there is some constant c for which $\mathcal{E} = \mathcal{O}(c) * \Delta_{\mathcal{R}} \Rightarrow \frac{\mathcal{E}}{\Delta_{\mathcal{R}}} = O(c)$.

Consider figure 6.5, it shows two graphs $\frac{\mathcal{E}}{\Delta_{\mathcal{R}}}$. The left one shows this ratio for $0 < |u| \leq 150$, and $n \in \{10, 100, 1000\}$. We see that $\frac{\mathcal{E}}{\Delta_{\mathcal{R}}}$ grows towards some $c \approx 9.6$.

However as both $T.Enum$ and $\Delta_{\mathcal{R}}$ do not directly depend on n, all these lines are roughly the same. Therefore in the right graph we look at only $n = 10$, for $30 \leq |u| \leq 600$. Then we are able to indeed verify that $\frac{\mathcal{E}}{\Delta_{\mathcal{R}}}$ grows towards some constant c, which in this case is thus approximately 9.6.

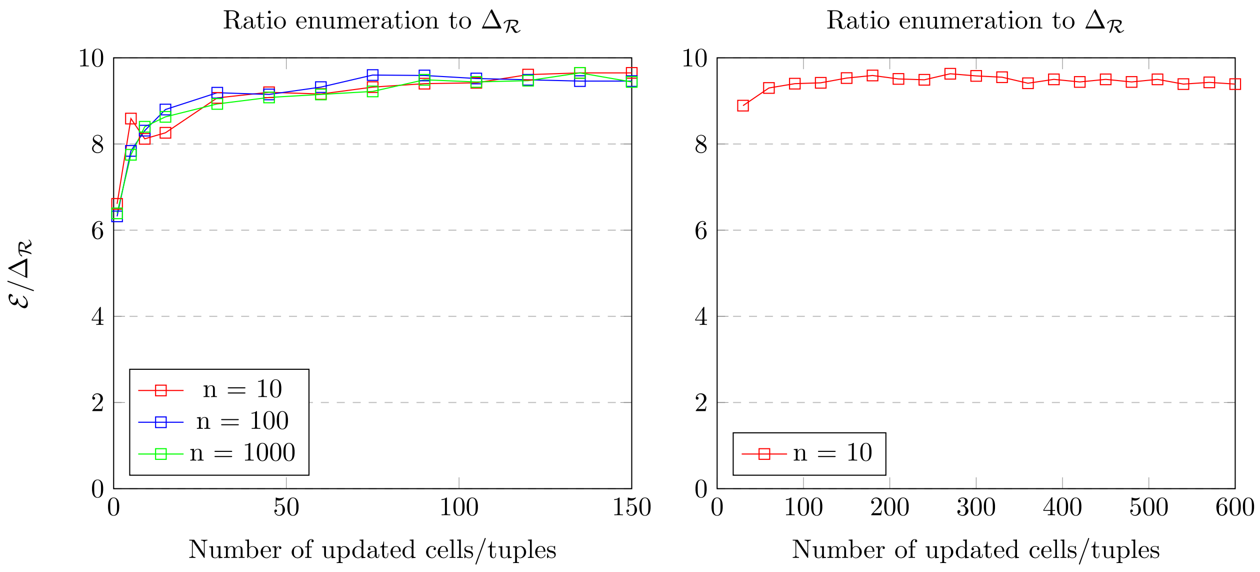


Figure 6.5: Ratio enumeration performance to input relation size per update size

Chapter 7

Conclusion

Our conclusion will be three-fold. First we will summarize the answers to the sub-research questions in order to finally answer the question *What does the Dynamic Yannakakis algorithm offer in terms of realizing IVM for Assemble models?*. Secondly we give recommendations to Anago, whether they should pursue Dynamic Yannakakis in order to find a method IVM or whether they should find their luck elsewhere. Thirdly we will state the contributions we made to the database research field in general.

7.1 Answering our research question

The state of Dynamic Yannakakis as given by the paper offers an algorithm such that the result of a query Q can be materialized and maintained efficiently.

Then the second sub-question, namely: 'Can we extend Dynamic Yannakakis to facilitate IVM for Assemble models, without nullifying the benefits of the current Dynamic Yannakakis algorithm?' we can answer with: yes we have extended Dynamic Yannakakis such that also a query Q containing aggregation, a combination of a join and aggregation, union, selection and anti-join operators can be materialized and also efficiently maintained using delta enumeration procedures. Then as the sub-models that we saw are convertable to one or more queries, which have as result the same relation as the sub-model this extended Dynamic Yannakakis algorithm facilitates IVM for Assemble models.

This is also verified by the results of experiments we conducted, which state that the delta enumeration procedure is efficient. Also the total update propagation procedure is efficient, which is the case because the number of tuples used only depends on the update size, rather than the total input relation size. Then still huge improvements can potentially be made when using a double-layered index, but for large dataset sizes and small update sizes, the current implementation - thus using only single-layered indices - is already more efficient than an equivalent Assemble model.

However this we may only conclude if we assume that a T-reduct consists of nodes that are augmented with relations ρ that are in the relation format and thus are sets of tuples.

We also want to answer the question: 'Considering the way Assemble is intertwined with the Oracle OLAP DML, would Dynamic Yannakakis be a practical short-term solution?'. Then as Assemble datasets are stored in Oracle databases, we have that for Dynamic Yannakakis to be a practical solution, it is not the case that relations ρ are sets of tuples, but rather multi-dimensional arrays. Additional to the challenges this produces for the described algorithms (especially the semi-join operations), these datasets also need to be stored and retrieved from an Oracle database. Thus in order to give a definitive answer to this question further research needs to be performed to see whether in the case where ρ relations are stored in Oracle databases, Dynamic Yannakakis is still efficient.

Concluding we can answer the question *What does the Dynamic Yannakakis algorithm offer in terms of realizing IVM for Assemble models?* in the following way: on long term, thus not assuming the limitations incurred by having to store data in an Oracle database Dynamic Yannakakis, at least for the Assemble model that we considered, facilitates efficient IVM for Assemble models.

In the short term, what it offers really depends on how the algorithm would perform if the augmented node relations are stored in an Oracle database.

7.2 Recommendations

From the answer on the research questions we can make the following recommendations. We have that Dynamic Yannakakis potentially does what it needs to do, namely efficiently maintain materialized output datasets. However whether this will work in practice depends on whether Oracle datasets can be integrated into Dynamic Yannakakis. Therefore it would be our recommendation to further investigate whether this is the case.

If this does not turn out to be the case, Dynamic Yannakakis should be given up on for now. Though, it could be reconsidered if no other satisfying solution for IVM has been found at the time Anago looks for a successor of the Oracle database for storing their data, where it could be considered to use a database application that uses relation like data structures for data storage.

If it does seem to be the case the next big problem is that of converting Assemble models into queries in an automized way. Thus we think investigating this problem should be a high priority as well.

7.3 Contributions

During this project we made the following contributions to research involving Dynamic Yannakakis:

- We extended Dynamic Yannakakis such that the output of queries containing aggregation, a combination of join and aggregation, selection, union and anti-join operators can be computed.
- Also we extended Dynamic Yannakakis such that updates to the output set can be computed efficiently, where the delta enumeration time complexity is linear in the size of Δ_{root_T} for a T-reduct T, which corresponds to a query Q.

Chapter 8

Future work

In this final chapter we will discuss how the research in this project can be continued. Here we will distinguish between general research on Dynamic Yannakakis, IVM using Dynamic Yannakakis in general and IVM for Assemble specifically. Note that research for Dynamic Yannakakis using IVM in general may also apply to IVM for Assemble.

8.1 Dynamic Yannakakis in general

For general research on Dynamic Yannakakis, we have that in this project we considered a number of query operators. These are join, anti-join, aggregation, join and aggregation, selection and union.

An operator that is missing in this list is the intersection operator. We did not consider this operator for the simple reason that no queries corresponding to an AC sub-model contained an intersection. Thus research could be done on extending Dynamic Yannakakis with the notion of an intersection-node.

Additionally for the anti-join node, it now only evaluates queries $A \triangleright B$, such that $var(A) \subseteq var(B)$. Thus additional research can be performed to alter the ComputeDelta algorithms for the anti-join node, such that $var(A) \subseteq var(B)$ does not necessarily have to hold.

For general research on using Dynamic Yannakakis facilitating IVM, research can be performed on how delta updates can be efficiently applied to relations and whether a double-layered index is a solution for this problem.

Another improvement could be made in having to materialize less output relations, by merging T-reductions. In general this can be done in the following way. Imagine there is a query $Q(\overline{A})$, where Q has a set of input relations $\overline{A}$. Then for each relation $R \in \overline{A}$, R it either is an atomic relation R_a or a derived relation R_d, which can be expressed by both an expression stating how the result can be derived from other relations and the name to which the result of this expression is saved. Then $Q(\overline{A})$ can be rewritten such that each R_d is not represented by the name, but by the expression. Then this process can recursively be repeated for input relations of each R_d, until all input relations are atomic relations.

However it is not certain that this query has one corresponding T-reduct. At least one property that a query can have, such that it does not have a corresponding T-reduct, is that it is cyclic as we saw with the moving-total query.

In order to thus minimize the number of materialized inbetween relations, research has to be performed on how a query - consisting of input queries which are derived - can be written out, before it is cyclic. This will be different for each query, but therefore it may be interesting to identify properties of queries that are cyclic, and also properties of queries that are not cyclic.

8.2 IVM for Assemble

Assuming that data should be stored in an Oracle database, research should be performed on how Dynamic Yannakakis can work if the augmented node relations ρ are datasets stored in an Oracle database, thus like multi-dimensional arrays, instead of the data structure considered during this project, namely relations with an hash-based index.

Then considering the conversion process of Anago models to T-reducts, an important step that has to be automized is the conversion of a sub-model to one or more queries. Therefore it needs to be investigated if such an automatic conversion process is possible and how it would work.

Additionally both in case where Dynamic Yannakakis turns out to be a method that facilitates IVM for Assemble and in case it does not, it may be worth looking further into the literature on IVM and dynamic query evaluation.

First of all, since we have that the Assemble's datasets are data cubes, Lee et al's work on IVM for data cubes may be very worth looking into.

However Assemble queries (corresponding to sub-models) and relations (corresponding to data sets), may share other properties, other than that the relations represent data cubes. Namely in this project we did not necessarily look for these properties as only a handful of queries were considered. Therefore it may be worth looking into whether such properties can be derived such that algorithms, aimed at more specific types of queries that coincide with Assemble sub-models may be considered. This because Dynamic Yannakakis is aimed at a rather general set of queries, which we made only more general by allowing additional query operators.

For example, since often sub-model results need to be combined in one data set, it might be worth looking into dynamic query evaluation for unions over queries.

Bibliography

[1] Anago - slimme planningsoplossingen. https://www.anago.nl/. (Accessed on 12/January/2022). 1

[2] Olap dml reference. https://docs.oracle.com/cd/E11882_01/olap.112/e17122/toc.htm, Nov 2013. (Accessed on 26/August/2022). 9

[3] Christoph Berkholz, Jens Keppeler, and Nicole Schweikardt. Answering ucqs under updates and in the presence of integrity constraints. *arXiv preprint arXiv:1709.10039*, 2017. 5

[4] Rada Chirkova, Jun Yang, et al. Materialized views. *Foundations and Trends in Databases*, 4(4):295–405, 2011. 4

[5] Muhammad Idris, Martín Ugarte, and Stijn Vansummeren. The dynamic yannakakis algorithm: Compact and efficient query processing under updates. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 1259–1274, 2017. 4

[6] Muhammad Idris, Martín Ugarte, Stijn Vansummeren, Hannes Voigt, and Wolfgang Lehner. Conjunctive queries with inequalities under updates. *Proceedings of the VLDB Endowment*, 11(7):733–745, 2018. 4

[7] Muhammad Idris, Martín Ugarte, Stijn Vansummeren, Hannes Voigt, and Wolfgang Lehner. Conjunctive queries with theta joins under updates. *arXiv preprint arXiv:1905.09848*, 2019. 4

[8] Muhammad Idris, Martín Ugarte, Stijn Vansummeren, Hannes Voigt, and Wolfgang Lehner. General dynamic yannakakis: conjunctive queries with theta joins under updates. *The VLDB Journal*, 29(2):619–653, 2020. 2

[9] Ahmet Kara, Hung Q Ngo, Milos Nikolic, Dan Olteanu, and Haozhe Zhang. Counting triangles under updates in worst-case optimal time. *arXiv preprint arXiv:1804.02780*, 2018. 5

[10] Ahmet Kara, Milos Nikolic, Dan Olteanu, and Haozhe Zhang. Trade-offs in static and dynamic evaluation of hierarchical queries. In *Proceedings of the 39th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, pages 375–392, 2020. 5

[11] Ki Yong Lee, Yon Dohn Chung, and Myoung Ho Kim. An efficient method for maintaining data cubes incrementally. *Information Sciences*, 180(6):928–948, 2010. 4

[12] Abraham Silberschatz, Henry Korth, and S. Sudarshan. *Database system concepts*. McGraw Hill education, 7th edition, 2020. 20

[13] Mihalis Yannakakis. Algorithms for acyclic database schemes. In *VLDB*, volume 81, pages 82–94, 1981. 4

Appendix A

Experiment results

no. of changed cells		**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
no. of employees	**10**	0	24 155	48 270	134 175	188 522	236 589	245 220	284 075	333 516	334 565
	100	0	29 759	62 545	168 828	193 690	208 779	318 736	327 226	394 428	477 094
	1000	0	31 906	61 055	178 510	201 102	226 910	344 395	366 502	374 133	475 313

Table A.1: Performance DYN $|u|$ <10

no. of changed cells		**15**	**30**	**45**	**60**	**75**	**90**	**105**	**120**	**135**	**150**
no. of employees	**10**	527 577	1 105 540	1 593 178	1 910 979	2 318 899	2 689 300	3 133 009	3 327 351	3 766 365	4 003 401
	100	784 430	1 511 640	2 091 033	2 818 383	3 224 515	3 788 466	4 484 111	4 723 099	5 226 534	5 503 234
	1000	797 987	1 449 242	2 020 530	2 594 871	3 219 691	3 982 654	4 123 675	5 028 278	5 335 596	5 886 073

Table A.2: Performance DYN $|u|$ >10

no. of changed cells		**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
no. of employees	**10**	0	2 554	4 868	11 657	11 417	17 678	24 402	24 386	28 765	34 549
	100	0	2 506	4 968	13 522	15 245	17 041	25 262	26 354	30 496	36 855
	1000	0	2 509	4 908	13 251	15 340	18 843	25 074	27 482	28 332	38 703

Table A.3: Performance DYN* $|u|$ <10

no. of changed cells		**15**	**30**	**45**	**60**	**75**	**90**	**105**	**120**	**135**	**150**
no. of employees	**10**	52 702	113 523	149 820	165 983	196 202	234 611	269 373	290 165	316 246	338 146
	100	61 129	113 639	153 199	201 912	235 985	270 804	304 282	318 085	347 965	366 365
	1000	62 253	108 781	148 538	183 881	222 044	272 461	287 412	335 071	367 573	381 121

Table A.4: Performance DYN* $|u|$ >10

Amount of employees		
10	**100**	**1000**
197 888	1 490 288	14 414 288

Table A.5: Assemble performance

no. of changed cells		**15**	**30**	**45**	**60**	**75**	**90**	**105**	**120**	**135**	**150**
no. of employees	**10**	2.67	5.59	8.05	9.66	11.72	13.59	15.83	16.81	19.03	20.23
	100	0.53	1.01	1.40	1.89	2.16	2.54	3.01	3.17	3.51	3.69
	1000	0.06	0.10	0.14	0.18	0.22	0.28	0.29	0.35	0.37	0.41

Table A.6: DYN / Assemble

no. of changed cells		**15**	**30**	**45**	**60**	**75**	**90**	**105**	**120**	**135**	**150**
no. of employees	**10**	0.27	0.57	0.76	0.84	0.99	1.19	1.36	1.47	1.60	1.71
	100	0.04	0.08	0.10	0.14	0.16	0.18	0.20	0.21	0.23	0.25
	1000	0.004	0.008	0.010	0.013	0.015	0.019	0.020	0.023	0.026	0.026

Table A.7: DYN* / Assemble

no. of changed cells		**15**	**30**	**45**	**60**	**75**	**90**	**105**	**120**	**135**	**150**
no. of employees	**10**	8.26	9.07	9.20	9.16	9.32	9.40	9.42	9.61	9.65	9.65
	100	8.80	9.19	9.15	9.32	9.60	9.59	9.52	9.49	9.46	9.46
	1000	8.63	8.93	9.08	9.15	9.22	9.49	9.44	9.47	9.65	9.44

Table A.8: Enumeration performance / delta root size

no. of changed cells	**30**	**60**	**90**	**120**	**150**	**180**	**210**	**240**	**270**	**300**
ratio	8.89	9.30	9.40	9.42	9.53	9.59	9.51	9.49	9.63	9.58
no. of changed cells	**330**	**360**	**390**	**420**	**450**	**480**	**510**	**540**	**570**	**600**
ratio	9.55	9.41	9.50	9.44	9.50	9.44	9.50	9.39	9.43	9.39

Table A.9: Enumeration performance / Delta root size, for n=10 and $|u|$ up until 600

Appendix B

Correctness proof of anti-join update and enumeration procedure

Proof update procedure of minus is correct
To prove:

$$\begin{aligned}
&\Delta_{true} = \Delta_{result}\\
&\equiv \Delta^+_{true} = \Delta^+_{result}\\
&\equiv \Delta^+_{true} = \Delta^+_{output} - \Delta^-_{output}\\
&\equiv \Delta^+_{true} \subseteq \Delta^+_{output} - \Delta^-_{output} \wedge \Delta^+_{output} - \Delta^-_{output} \subseteq \Delta^+_{true}
\end{aligned}$$

Part 1

$$\begin{aligned}
&\Delta^+_{output} - \Delta^-_{output} \subseteq \Delta^+_{true}\\
&\equiv \forall t' \in \Delta^+_{output} - \Delta^-_{output} \Rightarrow t' \in \Delta^+_{true}
\end{aligned}$$

Take an arbitrary tuple $t, t \in \Delta^+_{true}$.

$$\begin{aligned}
&t' \in \Delta^+_{true}\\
&\equiv t' \in (A - B)\\
&\equiv t' \in A \wedge B \ltimes t' = \emptyset
\end{aligned}$$

Thus we should prove that if a tuple t' is via algorithm 21 added to the result, thus added to Δ^+_{output} and not to Δ^-_{output}, then the expression:
$t' \in A \wedge B \ltimes t' = \emptyset$,
holds.

Consider the case where a tuple t' is added to Δ^+_p line 5 of algorithm 21. Since $t' \in \Delta^+_A$ and $\Delta^-_A = \emptyset$, $\underline{t' \in A}$ holds.

Case A is updated after B
Still consider line 5 of algorithm 21. Since B has already been updated and this line is reached, we have that $\underline{B \ltimes t' = \emptyset}$. Hence the expression holds regardless of whether or not t' is added to Δ^-_p. **Case A is updated before B**
At line 5 of of algorithm 21, since B at this point has not yet been updated, thus we have from

line 3 $\emptyset \ltimes t' = emptyset$.

Now consider case $n = B$. We want to prove that if t' is not added to Δ_p^- then $B \ltimes t' = \emptyset$. Assume that $B \ltimes t' \neq \emptyset$. Then there is a tuple $t2 \in B \ltimes t'$. Since B originally contains no tuples we therefore also have that $t2 \in \Delta_B^+ \ltimes t'$, and thus $t2 \in \Delta_B^+$.

In line 7 we therefore would iterate over $t2$ and since A at this point has been updated, $t' \in A$ and $t2'$ joins with t, t' will be found at line 8. Then, since $t' \neq NIL$ at line 10 it will be added to $Delta_p^-$.

Thus it follows from the assumption that $B \ltimes t' \neq \emptyset$ that t' will be added to $Delta_p^-$. This implies that if t' is not added to $Delta_p^-$, then $\underline{B \ltimes t' = \emptyset}$ must hold.

Hence, in both cases A is updated before B and vice versa, if t' is added to Δ_p^+ and not to $Delta_p^-$ then the expression holds.

Concluding: if $t' \in \Delta_p^+ \wedge t' \in \Delta_p^-$ and therefore also $t' \in \Delta_{output}^+ \wedge t' \in \Delta_{output}^-$, we have that $t' \in A \wedge B \ltimes t' = \emptyset$, which was what we needed to prove.

Part 2

$$\begin{aligned}&\Delta_{true}^+ \subseteq \Delta_{output}^+ - \Delta_{output}^-\\ &\equiv \forall t \in \Delta_{true}^+ \Rightarrow t \in \Delta_{output}^+ - \Delta_{output}^-\end{aligned}$$

Take an arbitrary tuple $t, t \in \Delta_{true}^+$.

$$\begin{aligned}&t \in \Delta_{true}^+\\ &\equiv t \in (A - B)\\ &\equiv t \in A \wedge B \ltimes t = \emptyset\end{aligned}$$

Hence we should prove that if for a tuple t, the boolean expression:
$t \in A \wedge B \ltimes t = \emptyset$
holds, then $t \in \Delta_{output}^+$ and $t \notin \Delta_{output}^-$.

If A is updated before B, then since $t \notin \emptyset \wedge t \in A$, we know that $t \in \Delta_A^+$. Thus in line 3, t is found. Then, since B has not been updated and $B = \emptyset$ intially line 4 will always return true. Thus, line 5 will be reached and t is added to Δ_p^+.

Then since $B \ltimes t = \emptyset$, a $t2 \in B \ltimes t$ does not exist which means that $t \leftarrow A \ltimes t2$ will never happen. Thus in this case t is not added to Δ_p^-.

If B is updated before A, then still since $t \notin \emptyset \wedge t \in A$, we know that $t \in \Delta_A^+$. Thus in line 3, t is found. Then since $B \ltimes t = \emptyset$, also line 5 will be reached and t is added to Δ_p^+.

In case $n = B$, A has not been updated, and since intially $A = \emptyset$, $A \ltimes t2$ will return NIL for any $t2$ and no tuple will be added to Δ_p^-, thus also not t for which $B \ltimes t = \emptyset$ holds.

Thus we have proven that if for a tuple t, it holds that $t \in A \wedge B \ltimes t = \emptyset$, then $t \in \Delta_p^+ \wedge t \in \Delta_p^-$ and therefore also $t \in \Delta_{output}^+ \wedge t \in \Delta_{output}^-$, which we needed to prove.

Conclusion
We have proven that $\Delta_{true}^+ \subseteq \Delta_{output}^+ - \Delta_{output}^-$ and $\Delta_{output}^+ - \Delta_{output}^- \subseteq \Delta_{true}^+$ and therefore we may conclude that $\Delta_{true} = \Delta_{result}$, which we needed to prove.

Algorithm 21 Update minus node initially

```
function ComputeDelta($n$)
    if $n = A$ then
        for each $t \in \Delta_A^+$ do
            if $B \ltimes t = \emptyset$ then
                $\Delta_p^+ += t$
    else
        for each $t2 \in \Delta_B^+$ do
            $t1 \leftarrow A \ltimes t2$
            if $t1 \neq NIL$ then
                $\Delta_p^- += \pi_{var(p)}(t1)$
```

To prove:

$$\begin{aligned}
&\Delta_{true} = \Delta_{result}\\
&\equiv \Delta_{true}^+ = \Delta_{result}^+ \wedge \Delta_{true}^- = \Delta_{result}^-\\
&\equiv \Delta_{true}^+ = \Delta_{output}^+ - \Delta_{output}^- \wedge \Delta_{true}^- = \Delta_{output}^- - \Delta_{output}^+\\
&\equiv \Delta_{true}^+ \subseteq \Delta_{output}^+ - \Delta_{output}^- \wedge \Delta_{output}^+ - \Delta_{output}^- \subseteq \Delta_{true}^+ \wedge\\
&\Delta_{true}^- \subseteq \Delta_{output}^- - \Delta_{output}^+ \wedge \Delta_{output}^- - \Delta_{output}^+ \subseteq \Delta_{true}^-
\end{aligned}$$

Part 1

$$\begin{aligned}
&\Delta_{output}^+ - \Delta_{output}^- \subseteq \Delta_{true}^+\\
&\equiv \forall t' \in \Delta_{output}^+ - \Delta_{output}^- \Rightarrow t' \in \Delta_{true}^+
\end{aligned}$$

Take an arbitrary tuple $t', t' \in \Delta_{true}^+$.

$$\begin{aligned}
&t' \in \Delta_{true}^+\\
&\equiv t' \in (A' - B') \wedge t' \notin (A - B)\\
&\equiv t' \in A' \wedge B' \ltimes t' = \emptyset \wedge (t' \notin A \vee B \ltimes t' \neq \emptyset)
\end{aligned}$$

Thus we should prove that if a tuple t' is via algorithm 22 added to the output, thus added to Δ_{output}^+ and not to Δ_{output}^- (and immediatly deleted), then the expression:
$t' \in A' \wedge B' \ltimes t' = \emptyset \wedge (t' \notin A \vee B \ltimes t' \neq \emptyset)$,
holds.

Case n = A
Consider algorithm 22, line 5. We will show that if t' is added to Δ_p^+ in case $n = A$ and not to Δ_p^-, then the expression holds.

At this point in the program we know that it holds for t' that $t' \in \Delta_A^+$, $t' \notin \Delta_A^-$. Then $t' \in \Delta_A^+ \wedge t' \notin \Delta_A^- \Rightarrow \underline{t' \in A'}$. Moreover we can deduce that $t' \notin A$:

For any arbitrary $t' \in \Delta_A^+$ we know that either: t' is an addition to A meaning that $t' \notin A$ inherently, or t' is an updated value. If t' is an updated value this means that there is a tuple t such that tRt' meaning that t is replaced by t', thus:

- $t \in \Delta_A^-$
- $t \in A$
- $t' \in A \Rightarrow t = t'$

Now we prove that $t' \notin A$. Assume that $t' \in A$, then $t = t'$. But then since $t \in \Delta_A^-$, $t' \in \Delta_A^-$ and we know that $t' \notin \Delta_A^-$, hence we arived to a contradiction and the assumption that $t' \in A$ was false. Thus indeed $\underline{t' \notin A}$.

Case A.1 A is updated after B
Since at line 5 in algorithm 22 B was already updated, we know that $\underline{B' \ltimes t = \emptyset}$. Thus in this case for t' the expression holds, regardless of whether t' was added to Δ_p^- or not.

Case A.2 A is updated before B
Now we prove that $B' \ltimes t' = \emptyset$ if t' is not added to Δ_p^- and A is updated before B.

First, since at line 5 in algorithm 22 B was not updated, we know that $B \ltimes t = \emptyset$

Assume that $B' \ltimes t' \neq \emptyset$. Consider line 7 of algorithm 23. We know that $B \ltimes t' = \emptyset$ and assuming that $B' \ltimes t' \neq \emptyset$, it means that there is a tuple $t2 \in B' \ltimes t$, $t2 \in \Delta_B^+ \ltimes t'$ and $t2 \notin \Delta_B^- \ltimes t'$. Then as $t2 \in B' \ltimes t'$, $t' = S \ltimes t2$ for any relation S, $t' \in S$. Thus since $t' \in A'$, $t' = A' \ltimes t2$ holds. Therefore, since A is updated t' would be found in line 8. Since $t2 \notin \Delta_B^- \ltimes t$ and $t' \neq NIL$ line 10 is reached.

Then a variable *count* is intialized to 0. At line 17 *count* $= |B' \ltimes t'|$ - $|\Delta_B^+ \ltimes t'| + |\Delta_B^+ \ltimes t'|$, which is equal to $|B \ltimes t'|$. As $|B \ltimes t'| = 0$, the final value of *count* is as well. Since *count* would thus be 0, t' would be added to Δ_p^-.

Hence it follows that if $B' \ltimes t' \neq \emptyset$ then t' is added to Δ_p^-. Therefore in the case that t' is not added to Δ_p^-, $\underline{B' \ltimes t' = \emptyset}$. Thus indeed in this case the expression holds.

Concluding: if t' is added in algorithm 22 line 5 to Δ_p^+ and not to Δ_p^- in algorithm 23, then both if A is updated before B and vice versa, the expression holds.

Case n = B
Consider that a tuple t' has been added to Δ_p^+ in line 10 of algorithm 22. We know that since B has been updated in this case, that $\underline{B' \ltimes t' = \emptyset}$.

We also know that there is a $t2$, such that $t' = A \ltimes t2$, $t2 \notin \Delta_B^+$ and $t2 \in \Delta_B^-$. $t2 \notin \Delta_B^+$ and $t2 \in \Delta_B^-$ means that $t2 \in B$. Since $t' = A \ltimes t2$ we know that $t2 \in S \ltimes t'$ for any relation S, $t2 \in S$, hence also for B. Therefore $t2 \in B \ltimes t'$ and $\underline{B \ltimes t' \neq \emptyset}$.

Case B.1 A is updated after B
We should prove that if t' is not added to Δ_p^-, then $t' \in A'$. Therefore assume that $t' \notin A'$. Then since B was updated before and the algorithm found t' in line 8 of algorithm 22, we know $t' \in A$. Thus since $t' \notin A'$ and $t' \in A$, we know that $t' \in \Delta_A^-$ and $t' \notin \Delta_A^+$.

Then in case $n = A$ of algorithm 23, following our assumption we have that $t' \in \Delta_A^-$, thus t' will be iterated over in line 3. Then since B already has been updated, $B' \ltimes t' = \emptyset$ and also $t' \notin \Delta_A^+$, t' would be added to Δ_p^-.

Hence, from our assumption that $t' \notin A'$, it follows that $t' \in \Delta_p^-$. However, $t' \notin \Delta_p^-$, hence $\underline{t' \in A'}$ has to hold.

Case B.2 A is updated before B
Now since A was already updated when in line 10 of algorithm 22 we added t' to Δ_A^+, we have that $t' \in A' \ltimes t2$ for at least one $t2$, thus $\underline{t' \in A}$. Hence in this case the expression holds for any t' that is added in line 10 to Δ_A^+, regardless of whether t' was added to Δ_p^- or not.

Concluding: if t' is added in 22, line 10 to Δ_p^+ and not to Δ_p^- in 23, then both if A is updated before B and vice versa, the expression holds.

Finally we may conclude that whenever a tuple t' is added to Δ_p^+ and not to Δ_p^-, and consequently - following algorithm EnumerateAdded() - t' is added to Δ_{output}^+ and not to Δ_{output}^-, then the expression:
$t' \in A' \wedge B' \ltimes t' = \emptyset \wedge (t' \notin A \vee B \ltimes t' \neq \emptyset)$,
holds, which was what we needed to prove.

Part 2

$$\Delta_{output}^- - \Delta_{output}^+ \subseteq \Delta_{true}^-$$
$$\equiv \forall t' \in \Delta_{output}^- - \Delta_{output}^+ \Rightarrow t' \in \Delta_{true}^-$$

Take an arbitrary tuple $t, t \in \Delta_{true}^-$.

$$t \in \Delta_{true}^-$$
$$\equiv t \notin (A' - B') \wedge t \in (A - B)$$
$$\equiv t \in A \wedge B \ltimes t = \emptyset \wedge (t \notin A' \vee B' \ltimes t \neq \emptyset)$$

Thus we should prove that if a tuple t is via algorithm 22 or 23 removed from the output, thus added to Δ_{output}^- and not also added to Δ_{output}^+, then the expression:
$t \in A \wedge B \ltimes t = \emptyset \wedge (t \notin A' \vee B' \ltimes t \neq \emptyset)$
holds.

Case n = A
Consider the case when a tuple t is added to Δ_p^- in line 5 of algorithm 23. We have that $t \in \Delta_A^-$ and $t \notin \Delta_A^+$. Therefore we may conclude that $\underline{t \in A}$ and $\underline{t \notin A'}$.

Case A.1 A is updated before B
We still consider line 5 of algorithm 23. Since B has not been updated then we know that $\underline{B \ltimes t = \emptyset}$. Hence in this case, regardless of whether or not t is added to Δ_p^+, we know that the boolean expression holds.

Case A.2 B is updated before A
Again consider line 5 of algorithm 23. Now, since B has already been updated at this point, we know that $B' \ltimes t = \emptyset$. In order to proof that $B \ltimes t = emptyset$ if t is not added to Δ_p^+, we consider algorithm 22, case $n = B$.

Assume that $B \ltimes t \neq emptyset$. Since $B' \ltimes t = \emptyset$ we have that there would exist a tuple $t2$, $t2 \in \Delta_B^- \ltimes t$, $t2 \notin \Delta_B^+ \ltimes t$. From $t2 \in \Delta_B^- \ltimes t$ it follows that $t2 \in \Delta_B^-$. Also it follows that $t2 \in S \ltimes t$ for any relation S, $t2 \in S$. Therefore, since $t2 \notin \Delta_B^+ \ltimes t$ it follows that $t2 \notin \Delta_B^+$.
Following our assumption, since $t2 \in \Delta_B^-$, t would be found at line 8. Then, since $t2 \notin \Delta_B^+$, $t \neq NIL$, B is updated and $B' \ltimes t = \emptyset$, t would at line 10 be added to Δ_p^+.

Hence, from our assumption that $B \ltimes t \neq \emptyset$, it follows that t will be added to Δ_p^+. Therefore we may conclude that in case t is not added to Δ_p^+ it must hold that $\underline{B \ltimes t = \emptyset}$. Thus also if B is updated before A the boolean expression holds.

Concluding: if t is added in algorithm 23 line 5 to Δ_p^- and is not added to Δ_p^+, then both if A is updated before B and vice versa, the expression holds.

Case n = B
Consider the case when a tuple $t1$ is added to Δ_p^- on line 18 of algorithm 23. We have that there exists a tuple $t2$, such that $t = A \ltimes t2$ (A being either A or A'), $t2 \in \Delta_B^+$ and $t2 \notin \Delta_B^-$. From $t2 \in \Delta_B^+$ and $t2 \notin \Delta_B^-$ it follows that $t2 \in B'$ and $t2 \notin B$. Because $t2 \in B'$ and t joins with $t2$, we know that $\underline{B' \ltimes t \neq \emptyset}$.

Since we know B has been updated and $|B' \ltimes t|$ - $|\Delta^+_B \ltimes t|$ + $|\Delta^-_B \ltimes t| = |B \ltimes t|$, we know that the value of *count* at line 18 is equal to $|B \ltimes t|$. Then since line 18 is reached, *count* $= 0$, we have that $|B \ltimes t| = 0$, hence $\underline{B \ltimes t = \emptyset}$.

Case B.1 A is updated before B
We still consider line 18 of algorithm 23. Then, since A has already been updated, we have that $t = A' \ltimes t2$ and therefore $t \in A'$.

Now consider case $n = A$ of algorithm 22. Assume that $t \notin A$. Then, from $t \in A'$ and $t \notin A$ it follows that $t \in \Delta^+_A$ and $t \notin \Delta^-_A$. Since at this point B has not been updated yet and $B \ltimes t = \emptyset$, we have that line 5 would be reached and t would be added to Δ^+_p.

Thus from the assumption that $tnotinA$ it follows that t will be added to Δ^+_p. This implies that if t is not added to Δ^+_p then $\underline{t \in A}$. Thus in case A is updated before B the boolean expression holds.

Case B.2 A is updated after B
From line 8 of algorithm 23. Since A has not yet been updated, we have that $t = A \ltimes t2$ and therefore $\underline{t \in A}$. Thus in this case the expression holds regardless of whether or not t is added to Δ^+_p.

Concluding: if t is added in algorithm 23 line 18 to Δ^-_p and is not added to Δ^+_p, then both if A is updated before B and vice versa, the expression holds.

Finally we may conclude that whenever a tuple t is added to Δ^-_p and not to Δ^+_p, and consequently - following algorithm EnumerateRemoved() - t is added to Δ^-_{output} and not to Δ^+_{output}, then the expression:
$t \in A \wedge B \ltimes t = \emptyset \wedge (t \notin A' \vee B' \ltimes t \neq \emptyset)$,
holds, which was what we needed to prove.

Part 3

$$\begin{aligned}&\Delta^+_{true} \subseteq \Delta^+_{output} - \Delta^-_{output}\\ &\equiv \forall t' \in \Delta^+_{true} \Rightarrow t' \in \Delta^+_{output} - \Delta^-_{output}\end{aligned}$$

Take an arbitrary tuple $t', t' \in \Delta^+_{true}$.

$$\begin{aligned}&t' \in \Delta^+_{true}\\ &\equiv t' \in (A' - B') \wedge t' \notin (A - B)\\ &\equiv t' \in A' \wedge B' \ltimes t' = \emptyset \wedge (t' \notin A \vee B \ltimes t' \neq \emptyset)\end{aligned}$$

Hence we must prove that if for a tuple t' it holds that:
$t' \in A' \wedge B' \ltimes t' = \emptyset \wedge (t' \notin A \vee B \ltimes t' \neq \emptyset)$,
then t' is added via algorithm 22 to Δ^+_p and not added to Δ^-_p via algorithm 23.

Case 1
$t' \in A' \wedge B' \ltimes t' = \emptyset \wedge t' \notin A$

Since $t' \in A'$ and $t' \notin A$, $t' \in \Delta^+_A$ and $t' \notin \Delta^-_A$ hold.

Then if B has been updated, since $B' \ltimes t' = \emptyset$ line 5 is reached and t' is added to Δ^+_p. Then also t' is not added via algorithm 23 to Δ^-_p, since in case $n = A$, $t' \notin \Delta^-_A$ and in case $n = B$, $t' \notin A$ hence $t' \notin A \ltimes t2'$ for any $t2' \in \Delta^+_B$.

If B has not been updated then there may be one or more $t2' \notin B' \ltimes t'$, but $t2' \in B \ltimes t'$, since $t2' \in \Delta_B^-$ and line 5 will not be reached. But then in case $n = A$, t' will be added: $t2' \in \Delta_B^-$, thus line 8 is reached. Then since A is updated, in line 8 t' is the tuple found by $A \ltimes t2'$. Then since $t2' \in \Delta_B^- \wedge t2' \notin B'$, $t2' \notin \Delta_B^+$ holds. Also t' is not NIL and since B now is updated and $B' \ltimes t' = \emptyset$ line 10 is reached and $t' \in \Delta_p^+$.

Also now t' is not added to Δ_p^-: in case $n = A$ we have established that $t' \notin \Delta_A^-$, so t' is not iterated over. In case $n = B$, there will be no $t2'$ such that $t' \in At2'$, because we know that since$t2' \notin B' \ltimes t'$ and $t2' \in B \ltimes t'$, surely $t2' \notin \Delta_B^+$. Thus indeed also in the case that B is updated after A, $t' \in \Delta_p^+$ and $t' \notin \Delta_p^-$.

Thus both when A is updated before B and B updated before A we have that $t' \in \Delta_p^+ \wedge t' \notin \Delta_p^-$, which we needed to prove.

Case 2
$t' \in A' \wedge B' \ltimes t' = \emptyset \wedge B \ltimes t' \neq \emptyset$

Then since $B' \ltimes t' = \emptyset \wedge B \ltimes t' \neq \emptyset$ we know there is at least one tuple $t2' \in B \ltimes t'$, $t2' \notin B' \ltimes t'$ and therefore $t2' \in \Delta_B^-$, $t2' \in B$ and $t2' \notin B'$. Moreover, for any tuple $t2' \in B \in t'$ these conditions hold.

If A has already been updated or $t' \in A$, then in case $n = B$ t' will be found in line 8 since then $t' = A \ltimes t2'$. Then since $t2' \in B$ and $t2' \notin B'$, $t2' \notin \Delta_B^+$. Also t' is not NIL we know that, in case $n = B$, B has been updated and since $B' \ltimes t' = \emptyset$ we reach line 10 and t' is added to Δ_p^+.

Then t' should not be added to Δ_p^-. In case $n = A$ of algorithm 23, since $t' \in A'$ we know that $t' \notin \Delta_A^-$ and will thus not be iterated over. In case $n = B$ we know that since for any $t2'$, $t2' \in B$ and $t2' \notin B'$ that $t2 \notin \Delta_B^+$, thus t' is never found in line 8. Hence if A has already ben updated or $t' \in A$ we have that $t' \in \Delta_p^+$ and $t' \notin \Delta_p^-$.

If A' has not been updated yet and $t' \notin A$, then since $t' \in A'$ we have that $t' \in \Delta_A^+$. Hence t' is iterated over in line 3 of case $n = A$. Then since $t' \in A'$, $t \notin \Delta_A^-$ holds. Also we know that since A in this case updates after B and $B' \ltimes t' = \emptyset$ line 5 will be reached and t' will be added to Δ_p^+.

Also now t' is not added to Δ_p^-. In case $n = A$, since $t' \in A'$ $t' \notin \Delta_A^-$ holds and t' will not be iterated over. In case $n = B$ since A has not been updated yet and $t' \notin A$, for no $t2 \in \Delta_B^+$ $t' = A \ltimes t2$, hence t' will not be added to Δ_p^-.

Thus also if B is updated before A and $t' \notin A$ we have that $t' \in \Delta_p^+$ and $t' \notin \Delta_p^-$.

Hence for any t', $t' \in A' \wedge B' \ltimes t' = \emptyset \wedge B \ltimes t' \neq \emptyset$ it holds that $t' \in \Delta_p^+$ and $t' \notin \Delta_p^-$, when A is updated before B and vice versa. This we needed to prove.

Concluding, we have proven that if for a tuple t' it holds that:
$t' \in A' \wedge B' \ltimes t' = \emptyset \wedge (t' \notin A \vee B \ltimes t' \neq \emptyset)$,
then t' is added via algorithm 22 to Δ_p^+ and not added to Δ_p^- via algorithm 23.

Part 4

$$\begin{aligned}&\Delta_{true}^- \subseteq \Delta_{output}^- - \Delta_{output}^+\\ &\equiv \forall t' \in \Delta_{true}^- \Rightarrow t' \in \Delta_{output}^- - \Delta_{output}^+\end{aligned}$$

Take an arbitrary tuple $t \in \Delta_{true}^-$.

$$\begin{aligned}&t \in \Delta_{true}^-\\ &\equiv t \notin (A' - B') \wedge t \in (A - B)\\ &\equiv t \in A \wedge B \ltimes t = \emptyset \wedge (t \notin A' \vee B' \ltimes t \neq \emptyset)\end{aligned}$$

Hence we must prove that if for a tuple t it holds that:
$t \in A \wedge B \ltimes t = \emptyset \wedge (t \notin A' \vee B' \ltimes t \neq \emptyset)$
then t is added via algorithm 23 to Δ_p^- and not to Δ_p^+ via algorithm 22.

Case 1
$t \in A \wedge B \ltimes t = \emptyset \wedge t \notin A'$

Since $t \in A \wedge t \notin A'$ we know that $t \in \Delta_A^-$. Also since $t \in A \wedge t \notin A'$, $t \notin \Delta_A^+$ holds.

If A is updated before B or $B' \ltimes t = \emptyset$, t is iterated over in case $n = A$ line 3 of algorithm 23 since $t \in \Delta_A^-$. $t \notin \Delta_A^+$ holds. Then since B has not yet been updated and $B \ltimes t = \emptyset$ or $B' \ltimes t = \emptyset$, we reach line 5 and $t \in \Delta_p^-$.

Also t is not added to Δ_p^+ via algorithm 22. In case $n = A$, since $t \notin \Delta_A^+$, t is not iterated over in line 3. In case $n = B$, since $B \ltimes t = \emptyset$ and $\Delta_B^- \subseteq B$, $\Delta_B^- \ltimes t = \emptyset$. Thus at line 8 t will not be found.

If B has been updated before A and $B' \ltimes t \neq \emptyset$, then since $B' \ltimes t \neq \emptyset$ and $B \ltimes t = \emptyset$, we have that there is at least a tuple $t2 \in \Delta_B^+ \ltimes t$, $t2 \notin \Delta_B^- \ltimes t$, $t2 \notin B \ltimes t$ and $t2 \in B' \ltimes t$. From $t2 \notin B \ltimes t$ and $t2 \in B'$ it follows that $t2 \notin B$. From $t2 \in B'$ and $t2 \notin B$ it follows that $t2 \notin \Delta^- B$. Note that since $\Delta_B^+ \ltimes t \subseteq \Delta_B^+$, $t2 \in \Delta_B^+$. Thus in case of line 7 of algorithm 23, $t2$ will be iterated over. Since A is not updated yet and $t \in A$, t will be found in line 8. Then since $t2 \notin \Delta_B^-$ and $t \neq NIL$, line 10 is reached.

Here a variable *count* is declared. $B \ltimes t = \emptyset$ we assumed and also $B' \ltimes t \neq \emptyset$. Therefore for any tuple $t2' \in B' \ltimes t \Rightarrow t2' \in \Delta_B^+ \ltimes t$. Also since $B \ltimes t = \emptyset$ we have that for any tuple $t2' \in \Delta_B^- \ltimes t \Rightarrow t2' \in \Delta_B^+ \ltimes t$. Then, because $\Delta_B^- \cup B' = \emptyset$ we have $|B' \ltimes t| + |\Delta_B^- \ltimes t| \leq |\Delta_B^+ \ltimes t|$. Vice versa for a tuple $t2' \in \Delta_B^+ \ltimes t$ it holds that (assuming no double tuples in Δ_A or Δ_B), $t2' \in B' \ltimes t \vee t2' \in \Delta_B^- \ltimes t$, thus $|\Delta_B^+ \ltimes t| \leq |B' \ltimes t| + |\Delta_B^- \ltimes t|$. Hence we may conclude $|\Delta_B^+ \ltimes t|$ $= |B' \ltimes t| + |\Delta_B^- \ltimes t|$. The final value of *count* will be $|B' \ltimes t| + |\Delta_B^- \ltimes t|$ - $|\Delta_B^+ \ltimes t|$ which will thus be 0. Therefore indeed t is added to $t \in \Delta_p^-$.

Finally we must show that $t \notin \Delta_p^+$ in this case. Consider algorithm 22. In case $n = A$, t will be not iterated over since $t \notin \Delta_A^+$. In case $n = B$, this also holds; since $B' \ltimes t \neq \emptyset$ and since B is updated, we know that if there is a tuple $t2' \in \Delta_B^-$, such that $t = A \ltimes t2'$, then $B \ltimes t$ is false and $t \notin \Delta+^p$.

Thus also when If B has been updated before A and $B' \ltimes t \neq \emptyset$ we have that $t \in \Delta-^p$ and $t \notin \Delta+^p$. Hence, when $t \in A \wedge B \ltimes t = \emptyset \wedge t \notin A'$ holds we know $t \in \Delta-^p$ and $t \notin \Delta+^p$, which is what we needed to prove.

Case 2
$t \in A \wedge B \ltimes t = \emptyset \wedge B' \ltimes t \neq \emptyset$

Since $B \ltimes t = \emptyset \wedge B' \ltimes t \neq \emptyset$ there is at least one tuple $t2 \in B' \ltimes t$, $t2 \notin B \ltimes t$, $t2 \in \Delta_B^+ \ltimes t$, $t2 \notin \Delta_B^- \ltimes t$. Moreover, since $t2 \in B' \ltimes t$ but $t2 \notin B \ltimes t$, we know that $t2 \notin B$, because if $t2 \in B$ and $t2 \in B' \ltimes t$ then $t2 \in B \ltimes t$, which is not the case. We may also conclude now that since $t2 \in B'$ and $t2 \notin B$ that $t2 \in \Delta_B^+$ and $t2 \notin \Delta_B^-$ hold. For now consider such tuple $t2$

If A has not been updated yet or $t \in A'$, we can consider case $n = B$ of algorithm 23. Then if $t \in A$ and A has not yet been updated, $t = A \ltimes t2$ thus t is found at line 8. If A has not been updated but $t \in A'$ t will also be found at line 8.

Then indeed $t2 \notin \Delta_B^-$ and t is not NIL hence again the algorithm will declare a *count* for which after the foreach loops the value will be 0, following the same argument in $Case1$ of $Part4$. Therefore t will be added to Δ_p^-.

Also we should prove that t will not be added to Δ_p^+. In case $n = A$ of algorithm 22, since B has been updated and $t2 \in B' \ltimes t$, if $t \in \Delta_A^+$ the algorithm will not reach line 5 since $B' \ltimes t \neq \emptyset$.

In case $n = B$ we know that line 10 will never be reached, since if t is found at line 8, then since B has been updated and $B' \ltimes t \neq \emptyset$, $B' \ltimes t = \emptyset$ is false.

Thus if B is updated before A, $t \in \Delta_p^-$ and $t \notin \Delta_p^+$.

If A has been updated before B, and $t \notin A'$, then since $t \in A$ it follows that $t \in \Delta_A^-$ and $t \notin \Delta_A^+$. Then because A is updated before B and $B \ltimes t$, line 5 is reached and t will be added to Δ_p^-.

Then in algorithm 22 in case $n = A$, $t \notin \Delta_A^+$ holds, thus t will not be iterated over and in case $n = B$ we have that since A is updated and $t \notin A'$, t wil not be found in line 8, thus also then t will not be added to Δ_p^+.

Thus also for any t, $t \in A \wedge B \ltimes t = \emptyset \wedge B' \ltimes t \neq \emptyset$ it holds that both when A is updated before B and vice versa, t is added to Δ_p^- and not to Δ_p^+. Thus in any case.

Now we finally may conclude that if for a tuple t it holds that:
$t \in A \wedge B \ltimes t = \emptyset \wedge (t \notin A' \vee B' \ltimes t \neq \emptyset)$
then t is added via algorithm 23 to Δ_p^- and not to Δ_p^+ via algorithm 22.

Conclusion
We have now proven that $\Delta_{true}^+ \subseteq \Delta_{output}^- - \Delta_{output}^+$ and $\Delta_{output}^+ - \Delta_{output}^- \subseteq \Delta_{true}^+$, from which follows that $\Delta_{output}^+ - \Delta_{output}^- = \Delta_{true}^+$ and moreover $\Delta_{output} = \Delta_{true}$, which we needed to prove.

Algorithm 22 Adding tuples to Δ_{output}^+

```
1: function ComputeDeltaPlus($n$)
2:     if $n = A$ then
3:         for each $t \in \Delta_A^+$ do
4:             if $t \notin \Delta_A^- \wedge B \ltimes t = \emptyset$ then
5:                 $\Delta_p^+ += t$
6:     else
7:         for each $t2 \in \Delta_B^-$ do
8:             $t1 \leftarrow A \ltimes t2$
9:             if $t2 \notin \Delta_B^+ \wedge t1 \neq NIL \wedge B \ltimes t1 = \emptyset$ then
10:                $\Delta_p^+ += \pi_{var(p)}(t1)$
11:
12: function EnumAdded($t_{parent}$)
13:     for each $t \in (\Delta_n^+ - \Delta_n^-) \ltimes t_{parent}$ do
14:         yield $t$
```

Algorithm 23 Adding tuples to Δ^-_{output}

```
function ComputeDeltaMinus($n$)
    if $n = A$ then
        for each $t \in \Delta^-_A$ do
            if $t \notin \Delta^+_A \wedge B \ltimes t = \emptyset$ then
                $\Delta^-_p += \pi_{var(p)}(t)$
    else
        for each $t2 \in \Delta^+_B$ do
            $t1 \leftarrow A \ltimes t2$
            if $t2 \notin \Delta^-_B \wedge t1 \neq NIL$ then
                $count \leftarrow 0$
                for each $t2' \in B \ltimes t1$ do
                    $count += t2'.count$
                for each $t2' \in \Delta^+_B \ltimes t1$ do
                    $count -= t2'.count$
                for each $t2' \in \Delta^-_B \ltimes t1$ do
                    $count += t2'.count$
                if $count == 0$ then
                    $\Delta^-_p += t1$

function EnumRemoved($t_{parent}$)
    for each $t \in (\Delta^-_n - \Delta^+_n) \ltimes t_{parent}$ do
        yield $t$
```

Appendix C

Full update analysis AC model

Update 1

Update: *<DC_001(Employee 'a', Fact 'holiday', Week '$w_{b.y}$'), x>*
This update is applied to datacube *001 Hours per fact per employee per week* and changes the cell at Employee a, Fact $Holiday$ and Week b to integer x.
2 - Compute net availability per employee per week
Definitions

- Let c be *DC_001(Employee 'a', Fact 'gross availability', Week '$w_{b.y}$')*
- Let d be *DC_001(Employee 'a', Fact 'education', Week '$w_{b.y}$')*
- Let e be the previous value of *DC_001(Employee 'a', Fact 'net availability', Week '$w_{b.y}$')*

Update methods

1. · *<DC_001(Employee 'a', Fact 'net availability', Week '$w_{b.y}$'), c - x - d>*
2. · *<DC_001(Employee 'a', Fact 'net availability', Week '$w_{b.y}$'), e - x + x'>*

Analytics

1. Input cells: 3, Updated cells: 1
2. Input cells: 3, Updated cells: 1

3 - Rollup hours per employee over weeks to year
Definitions

- Let $c_{i,f}$ be *DC_001(Employee 'a', Fact 'f', Week '$w_{i,y}$')*
- Let d_f be the previous value of *DC_001(Employee 'a', Fact 'f', Week 'y')*

Update methods

1. · *<DC_001(Employee 'a', Fact 'holiday', Week 'y'),* $\sum_{1 \leq i \leq len(y)} c_{i,holiday}$ *>*
   · *<DC_001(Employee 'a', Fact 'net availability', Week 'y'),* $\sum_{1 \leq i \leq len(y)} c_{i,netav.}$*>*
2. · *<DC_001(Employee 'a', Fact 'net availability', Week 'y'),* $d_{netav.}$ *- x + x'>*
   · *<DC_001(Employee 'a', Fact 'holiday', Week 'y'),* $d_{holiday}$ *+ x - x'>*

Analytics

1. Input cells: 2 * $len(y)$, Updated cells: 2
2. Input cells: 4, Updated cells: 2

4 - Fill in hours where employee and week correspond to team
Definitions

- Let c be *DC_001(Employee 'a', Fact 'net availability', Week '$w_{b.y}$')*
- Let d_f be the previous value of *DC_003(Employee 'a', Fact 'f', Team 't', Week '$w_{b.y}$')*

Update methods

1. · *<DC_003(Employee 'a', Fact 'net availability', Team 't', Week '$w_{b.y}$'), c >*
   · *<DC_003(Employee 'a', Fact 'holiday', Team 't', Week '$w_{b.y}$'), x >*
2. · *<DC_003(Employee 'a', Fact 'net availability', Team 't', Week '$w_{b.y}$'), $d_{netav.}$ - x + x'>*
   · *<DC_003(Employee 'a', Fact 'holiday', Team 't', Week '$w_{b.y}$'), x>*

Analytics

1. Input cells: 2, Updated cells: 2
2. Input cells: 3, Updated cells: 2

5 - Sum hours over employees to team (per week)
Definitions

- Let $c_{i,f}$ be *DC_003(Employee 'i', Fact 'f', Team 't', Week '$w_{b.y}$')*
- Let d_f be the previous value of *DC_002(Fact 'f', Team 't', Week '$w_{b.y}$')*

Update methods

1. · *<DC_002(Fact 'net availability', Team 't', Week '$w_{b.y}$'), $\sum\limits_{i \in set(t)} c_{i,netav.}$ >*
   · *<DC_002(Fact 'holiday', Team 't', Week '$w_{b.y}$'), $\sum\limits_{i \in set(t)} c_{i,holiday}$ >*
2. · *<DC_002(Fact 'net availability', Team 't', Week '$w_{b.y}$'), $d_{netav.}$ - x + x'>*
   · *<DC_002(Fact 'holiday', Team 't', Week '$w_{b.y}$'), $d_{holiday}$ + x - x'>*

Analytics

1. Input cells: 2 * $|set(t)|$, Updated cells: 2
2. Input cells: 4, Updated cells: 2

6 - Rollup hours over team to levels department and total
Definitions

- Let $c_{i,f}$ be *DC_002(Fact 'f', Team 'i', Week '$w_{b.y}$')*
- Let $d_{i,f}$ be the previous value of *DC_002(Fact 'f', Team 'i', Week '$w_{b.y}$')*

Update methods

1. For fact $f \in$ {*holiday, net availability*} and $i \in \{D, T\}$:

· *<DC_002(Fact 'f', Team 'i', Week '$w_{b.y}$'),* $\sum_{j \in set(i)} c_{j,f}$ *>*

2. For team $i \in \{D,\ T\}$:

   · *<DC_002(Fact 'net availability', Team 'i', Week '$w_{b.y}$'),* $d_{i,netav.}$ *- x + x' >*

   · *<DC_002(Fact 'holiday', Team 'i', Week '$w_{b.y}$'),* $d_{i,holiday}$ *+ x - x' >*

Analytics

1. Input cells: 2 * ($|set(D)| + |set(T)|$), Updated cells: 4
2. Input cells: 4, Updated cells: 4

7 - Compute adjusted hours
Definitions

- Let $c_{i,f}$ be *DC_002(Fact 'f', Team 'i', Week '$w_{b.y}$')*
- Let d_f be *DC_006(Fact 'f', Week '$w_{b.y}$')*
- Let $e_{i,f}$ be the previous value of *<DC_007(Fact 'f', Team 'i', Week '$w_{b.y}$')*

Update methods

1. For fact $f \in \{$*holiday, net availability*$\}$ and team $i \in \{t, D, T\}$:

   · *<DC_007(Fact 'f', Team 'i', Week '$w_{b.y}$'),* $c_{i,f}$ ** (1 +* d_f*)>*

2. For team $i \in \{t, D, T\}$:

   · *<DC_007(Fact 'net availability', Team 'i', Week '$w_{b.y}$'),*
   $e_{i,netav.}$ *+ (-x + x') * (1 +* $d_{netav.}$*)>*

   · *<DC_007(Fact 'holiday', Team 'i', Week '$w_{b.y}$'),* $e_{i,holiday}$ *+ (x - x') * (1 +* $d_{holiday}$*)>*

Analytics

1. Input cells: 8, Updated cells: 6
2. Input cells: 10, Updated cells: 6

8 - Compute moving total
Definitions

- Let $c_{i,j,f,l}$ be *DC_002(Fact 'f', Team 'i', Week '$w_{j.l}$')*
- Let $d_{i,j,f,l}$ be the previous value of *DC_004(Fact 'f', Team 'i', Week '$w_{j.l}$')*

For the given definitions it may occur that $j > len(l)$. Therefore let in that case $w_{j.l} = w_{j-len(l),l+1}$, $c_{i,j,f,l} = c_{i,j-len(l),f,l+1}$ and $d_{i,j,f,l} = d_{i,j-len(l),f,l+1}$ hold.

Update methods

1. For fact $f \in \{$*holiday, net availability*$\}$, team $i \in \{t, D, T\}$ and $j \in [b, b+4]$:

   · *<DC_004(Fact 'f', Team 'i', Week '$w_{j.y}$'),* $\sum_{k=j-4}^{j} c_{i,k,f,y}$*>*

2. For team $i \in \{t, D, T\}$ and $j \in [b, b+4]$:

· *<DC_004(Fact 'net availability', Team 'i', Week '$w_{j.y}$'), $d_{i,j,netav.,y} - x + x'$>*
· *<DC_004(Fact 'holiday', Team 'i', Week '$w_{j.y}$'), $d_{i,j,holiday,y}$ + x - x'>*

Analytics

1. Input cells: 54, Updated cells: 30
2. Input cells: 32, Updated cells: 30

9 - Rollup hours per team over week to year
Definitions

- Let $c_{i,j,f}$ be *DC_002(Fact 'f', Team 'i', Week '$w_{j,y}$')*
- Let $d_{i,f}$ be the previous value of *DC_002(Fact 'f', Team 'i', Week 'y')*

Update methods

1. For fact $f \in$ {*holiday, net availability*} and team $i \in \{t, D, T\}$:
   · *<DC_001(Fact 'f', Team 'i', Week 'y'), $\sum_{1 \leq i \leq len(y)} c_{i,j,f}$ >*
2. For team $i \in \{t, D, T\}$:
   · *<DC_001(Fact 'net availability', Team 'i', Week 'y'), $d_{i,netav.}$ - x + x'>*
   · *<DC_001(Fact 'holiday', Team 'i', Week 'y'), $d_{i,holiday}$ + x - x'>*

Analytics

1. Input cells: 6 * $len(y)$, Updated cells: 6
2. Input cells: 8, Updated cells: 6

10 - Rollup adjusted hours per team over week to year
Definitions

- Let $c_{i,j,f}$ be *DC_007(Fact 'f', Team 'i', Week '$w_{j.y}$')*
- Let $d_{i,f}$ be the previous value of *DC_007(Fact 'f', Team 'i', Week 'y')*
- Let $e_{i,f}$ be *DC_006(Fact 'f', Week 'y')*

Update methods

1. For fact $f \in$ {*holiday, net availability*} and team $i \in \{t, D, T\}$:
   · *<DC_007(Fact 'f', Team 'i', Week 'y'), $\sum_{1 \leq i \leq len(y)} c_{i,j,f}$ >*
2. For team $i \in \{t, D, T\}$:
   · *<DC_007(Fact 'net availability', Team 'i', Week 'y'), $d_{i,netav.}$ + (-x + x') * (1 + $e_{i,netav.}$) >*
   · *<DC_007(Fact 'holiday', Team 'i', Week 'y'), $d_{i,holiday}$ + (x - x') * (1 + $e_{i,holiday}$) >*

Analytics

1. Input cells: 6 * $len(y)$, Updated cells: 6
2. Input cells: 14, Updated cells: 6

11 - Compute percentage facts moving total
Definitions

- Let $c_{i,j,f,l}$ be *DC_004(Fact 'f', Team 'i', Week '$w_{j.l}$')*
- Let $d_{i,j,l}$ be the previous value of *DC_008(Percentage fact '%holiday', Team 'i', Week '$w_{j.l}$')*

For the given definitions it may occur that $j > len(l)$. Therefore let in that case $w_{j.l} = w_{j-len(l),l+1}$, $c_{i,j,f,l} = c_{i,j-len(l),f,l+1}$ and $d_{i,j,l} = d_{i,j-len(l),l+1}$ hold.

Update methods

1. For team $i \in \{t, D, T\}$ and $j \in [b, b+4]$:

   · *<DC_008(Percentage fact '%holiday', Team 'i', Week '$w_{j.y}$'), $\frac{c_{i,j,holiday,y}}{c_{i,j,grossav.,y}}$ >*

2. For team $i \in \{t, D, T\}$ and $j \in [b, b+4]$:

   · *<DC_008(Percentage fact '%holiday', Team 'i', Week '$w_{j.y}$'), $d_{i,j,y}$ + $\frac{x-x'}{c_{i,j,grossav.,y}}$ >*

Analytics

1. Input cells: 30, Updated cells: 15
2. Input cells: 32, Updated cells: 15

Update 2

Update: *<R_001(Employee 'a', Week $w_{b.y}$), Team 't'>* This update is applied to relation *001 Employee x Week - Team* and changes the cell at Employee a and Week $w_{b.y}$ to Team t.

Autofill
As this relation has the property 'Autofill relation based on time', all cells at Employee a and from Week $b+1$ onwards will be changed to t. *Update methods*

1. For week $i \in future\_weeks$

   · *<R_001(Employee 'a', Week 'i'), t>*

Analytics

1. Input cells: 1, Updated cells: $|future\_weeks|$

4 - Fill in hours where employee and week correspond to team
Definitions

- Let $c_{i,f}$ be *DC_001(Employee 'a', Fact 'f', week 'i')*

Update methods
For fact $f \in$ {*gross availability, holiday, education, net availability*} and week $i \in future\_weeks^+$:

· *<DC_003(Employee 'a', Fact 'f', Team 't', Week 'i'), $c_{i,f}$ >*

· *<DC_003(Employee 'a', Fact 'f', Team 't'_i', Week 'i'), na>*

Analytics
Input cells: 4 * $|future\_weeks^{+}|$, Updated cells: 8 * $|future\_weeks^{+}|$

5 - Sum hours over employees to team (per week)
Definitions

- Let $c_{i,f,j,k}$ be *DC_003(Employee 'i', Fact 'f', Team 'j', Week 'k')*
- Let $d_{f,i,j}$ be the previous value of *DC_002(Fact 'f', Team 'i', Week 'j')*
- Let $e_{f,i}$ *DC_001(Employee 'a', Fact 'f', Week 'i')*

Update methods
For fact $f \in \{$*gross availability, holiday, education, net availability*$\}$, week $i \in future\_weeks^{+}$:

1. · *<DC_002(Fact 'f', Team 't', Week 'i'),* $\sum\limits_{j \in set(t)} c_{j,f,t,i}$ *>*
   · *<DC_002(Fact 'f', Team '*t'_i*', Week 'i'),* $\sum\limits_{j \in set(t'_i)} c_{j,f,t'_i,i}$ *>*
2. · *<DC_002(Fact 'f', Team 't', Week 'i'),* $d_{f,t,i}$ + $e_{f,i}$ *>*
   · *<DC_002(Fact 'f', Team '*t'_i*', Week 'i'),* $d_{f,t'_i,i}$ - $e_{f,i}$ *>*

Analytics

1. Input cells: 4 * $\sum\limits_{i \in future\_weeks^{+}} (\sum\limits_{j \in set(t)} (1) + \sum\limits_{j \in set(t'_i)} (1)) \approx 8 * |future\_weeks^{+}| * |set(t)|$,
   Updated cells: 8 * $|future\_weeks^{+}|$
2. Input cells: 12 * $|future\_weeks^{+}|$, Updated cells: 8 * $|future\_weeks^{+}|$

6 - Rollup hours over team to level department (total does not change)
Definitions

- Let $c_{f,i,j}$ be *DC_002(Fact 'f', Team 'i', Week 'j')*
- Let $d_{f,i,j}$ be the previous value of *DC_002(Fact 'f', Team 'i', Week 'j')*
- Let $e_{f,i}$ be *DC_001(Employee 'a', Fact 'f', Week 'i')*

Update methods

1. For fact $f \in \{$*gross availability, holiday, education, net availability*$\}$, $i \in \{D, D'_j\}$ and week $j \in future\_weeks^{+}$:
   · *<DC_002(Fact 'f', Team 'i', Week 'j'),* $\sum\limits_{k \in set(i)} c_{k,f,j}$ *>*
2. For fact $f \in \{$*gross availability, holiday, education, net availability*$\}$ and week $j \in future\_weeks^{+}$:
   · *<DC_002(Fact 'f', Team 'D', Week 'j'),* $d_{f,D,j}$ + $e_{f,i}$ *>*
   · *<DC_002(Fact 'f', Team '*D'_j*', Week 'j'),* $d_{f,D'_j,j}$ - $e_{f,i}$ *>*

Analytics

1. Input cells: $\approx$ 8 * $|future\_weeks^{+}|$ * $|set(D)|$, Updated cells: 8 * $|future\_weeks^{+}|$
2. Input cells: 12 * $|future\_weeks^{+}|$, Updated cells: 8 * $|future\_weeks^{+}|$

7 - Compute adjusted hours
Definitions

- Let $c_{f,i,j}$ be *DC_002(Fact 'f', Team 'i', Week 'j')*
- Let $d_{f,j}$ be *DC_006(Fact 'f', Week 'j')*
- Let $e_{f,i,j}$ be the previous value of *<DC_007(Fact 'f', Team 'i', Week 'j')*
- Let $g_{f,i}$ be *DC_001(Employee 'a', Fact 'f', Week 'i')*

Update methods

1. For fact $f \in \{$*gross availability, holiday, education, net availability*$\}$, team $i \in \{t, D, t'_j D'_j\}$ and $j \in future\_weeks^+$:
   - *<DC_007(Fact 'f', Team 'i', Week 'j'),* $c_{f,i,j}$ ** (1 +* $d_{f,j}$*)>*
2. For fact $f \in \{$*gross availability, holiday, education, net availability*$\}$, team $i \in \{t, D\}$, team $k \in \{t'_j D'_j\}$ and $j \in future\_weeks^+$:
   - *<DC_007(Fact 'f', Team 'i', Week 'j'),* $e_{f,i,j}$ *+* $g_{f,i}$ ** (1 +* $d_{f,j}$*)>*
   - *<DC_007(Fact 'f', Team 'k', Week 'j'),* $e_{f,k,j}$ *-* $g_{f,i}$ ** (1 +* $d_{f,j}$*)>*

Analytics

1. Input cells: 20 * $|future\_weeks^+|$, Updated cells: 16 * $|future\_weeks^+|$
2. Input cells: 24 * $|future\_weeks^+|$, Updated cells: 16 * $|future\_weeks^+|$

8 - Compute moving total
Definitions

- Let $c_{f,i,j}$ be *DC_002(Fact 'f', Team 'i', Week 'j')*
- Let $d_{f,i,j}$ be the previous value of *DC_004(Fact 'f', Team 'i', Week 'j')*
- Let $e_{f,i}$ be *DC_001(Employee 'a', Fact 'f', Week 'i')*

Update methods

1. For fact $f \in \{$*gross availability, holiday, education, net availability*$\}$, team $i \in \{t, D, t'_j D'_j\}$ and $j \in future\_weeks^+$:
   - *<DC_004(Fact 'f', Team 'i', Week 'j'),* $\sum_{k=j-4}^{j} c_{f,i,k}$*>*
2. For fact $f \in \{$*gross availability, holiday, education, net availability*$\}$, team $i \in \{t, D\}$, team $k \in \{t'_j D'_j\}$ and $j \in future\_weeks^+$:
   - *<DC_004(Fact 'f', Team 'i', Week 'j'),* $d_{f,i,j} + \sum_{l=j-4}^{j} e_{f,l}$*>*
   - *<DC_004(Fact 'f', Team 'k', Week 'j'),* $d_{f,k,j} - \sum_{l=j-4}^{j} e_{f,l}$*>*

Analytics

1. Input cells: 16 * $|future\_weeks^+|$ + 64, Updated cells: 16 * $|future\_weeks^+|$

2. Input cells: 20 * $|future\_weeks^{+}|$ + 16, Updated cells: 16 * $|future\_weeks^{+}|$

9 - Rollup hours per team over week to year
Definitions

- Let $c_{f,i,j,k}$ be *DC_002(Fact 'f', Team 'i', Week '$w_{j.k}$')*
- Let $d_{f,i,j}$ be the previous value of *DC_002(Fact 'f', Team 'i', Week 'j')*
- Let $e_{f,i}$ be *DC_001(Employee 'a', Fact 'f', Week 'i')*

Update methods

1. For fact $f \in \{$*gross availability, holiday, education, net availability*$\}$, team $i \in \{t, D, t'_j, D'_j\}$ and week $j \in \{y, y+1\}$:
   - *<DC_001(Fact 'f', Team 'i', Week 'j'),* $\sum_{1 \leq k \leq len(j)} c_{f,i,k,j}$ *>*
2. For fact $f \in \{$*gross availability, holiday, education, net availability*$\}$, team $i \in \{t, D\}$, team $l \in \{t'_j, D'_j\}$ and week $j \in \{y, y+1\}$:
   - *<DC_001(Fact 'f', Team 'i', Week 'j'),* $d_{f,i,j}$ + $e_{f,j}$*>*
   - *<DC_001(Fact 'f', Team 'l', Week 'j'),* $d_{f,l,j}$ - $e_{f,j}$*>*

Analytics

1. Input cells: 16 * ($len(y)$ + $len(y+1)$), Updated cells: 32
2. Input cells: 48, Updated cells: 32

10 - Rollup adjusted hours per team over week to year
Definitions

- Let $c_{f,i,j,k}$ be *DC_007(Fact 'f', Team 'i', Week '$w_{j.k}$')*
- Let $d_{f,i,j}$ be the previous value of *DC_007(Fact 'f', Team 'i', Week 'j')*
- Let $e_{f,i,j}$ be *DC_006(Fact 'f', Week '$w_{i.j}$')*
- Let $g_{f,i,j}$ be *DC_001(Employee 'a', Fact 'f', Week '$w_{i.j}$')*

Update methods

1. For fact $f \in \{$*gross availability, holiday, education, net availability*$\}$, team $i \in \{t, D, t'_j, D'_j\}$ and week $j \in \{y, y+1\}$:
   - *<DC_007(Fact 'f', Team 'i', Week 'j'),* $\sum_{1 \leq k \leq len(j)} c_{f,i,k,j}$ *>*
2. For fact $f \in \{$*gross availability, holiday, education, net availability*$\}$, team $i \in \{t, D\}$, team $l \in \{t'_j, D'_j\}$ and week $j \in \{y, y+1\}$:
   - *<DC_007(Fact 'f', Team 'i', Week 'j'),* $d_{f,i,j}$ + $\sum_{1 \leq k \leq len(j)} (g_{f,k,j} * (1 + e_{f,k,j}))$ *>*
   - *<DC_007(Fact 'f', Team 'l', Week 'j'),* $d_{f,l,j}$ - $\sum_{1 \leq k \leq len(j)} (g_{f,k,j} * (1 + e_{f,k,j}))$ *>*

Analytics

1. Input cells: 16 * ($len(y)$ + $len(y+1)$), Updated cells: 32

2. Input cells: 32 + 16 * ($len(y)$ + $len(y+1)$), Updated cells: 32

11 - Compute percentage facts moving total
Definitions

- Let $c_{f,i,j}$ be *DC_004(Fact 'f', Team 'i', Week 'j')*

Update methods

1. For team $i \in \{t, D, t'_j, D'_j\}$ and $j \in future\_weeks$:

   · *<DC_008(Percentage fact '%holiday', Team 'i', Week 'j'),* $\frac{c_{holiday,i,j}}{c_{grossav.,i,j}}$ >
   · *<DC_008(Percentage fact '%education', Team 'i', Week 'j'),* $\frac{c_{education,i,j}}{c_{grossav.,i,j}}$ >

Analytics

1. Input cells: 30, Updated cells: 15

12 - Copy team per employee per week
Update methods

1. For week $i \in future\_weeks^+$:

   · *<DC_010(Employee 'a', Team 't', Week 'i'), true>*
   · *<DC_010(Employee 'a', Team 't'', Week 'i'), na>*

Analytics

1. Input cells: 0, Updated cells: 2 * $|future\_weeks^+|$

13 - Count amount of employees per team per week
Definitions

- Let $c_{i,j}$ be *DC_010(Employee 'a', Team 'i', Week 'j')*
- Let $d_{i,j}$ be the previous value of *DC_009(Team 'i', Week 'j')*

Update methods

1. For week $i \in future\_weeks^+$, $j \in \{t, t'\}$:

   · *<DC_009(Employee 'a', Team 'j', Week 'i'),* $\sum_{k \in set(j)}$ *1>*

2. For week $i \in future\_weeks^+$:

   · *<DC_009(Employee 'a', Team 't', Week 'i'),* $d_{t,i}$ *+ 1>*
   · *<DC_009(Employee 'a', Team 't'', Week 'i'),* $d_{t',i}$ *- 1>*

Analytics

1. Input cells: ($|future\_weeks^+|$ * $|set(t)|$) + ($|future\_weeks^+|$ * $|set(t')|$)
   Updated cells: 2 * $|future\_weeks^+|$

2. Input cells: 2 * $|future\_weeks^+|$, Updated cells: 2 * $|future\_weeks^+|$

14 - Determine empty teams per week
Definitions

- Let $c_{i,j,k}$ be *DC_010(Employee 'i', Team 'j', Week 'k')*
- Let $d_{i,j}$ be the previous value of *DC_011(Team 'i', Week 'j')*

Update methods

1. For week $i \in future\_weeks^{+}$:

    · *<DC_009(Employee 'a', Team 't', Week 'i'), na>*

    · *<DC_009(Employee 'a', Team 't′', Week 'i'), (* $\sum_{j \in set(t')}$ *(1)≥ 1)>*

Analytics

1. Input cells: $|future\_weeks^{+}|$ * $|set(t')|$, Updated cells: 2 * $|future\_weeks^{+}|$

Appendix D

Derivations AC sub-models to queries

D.1 Relational algebra

D.1.1 RM_BM

1 - Copy gross availability from DC001 to DC001
Relation to update: $DC001$*(employee, fact, week, hours)*
Example of relation:

Employee	Week	Fact	Hours
Rik	w12.2022	holiday	12
Rik	w12.2022	education	8

Input relation: $DC005$*(employee, week, gross availability)*
Example of relation:

Employee	Week	gross availability
Rik	w12.2022	40

Then the following tuple should be inserted into $DC001$

Employee	Week	Fact	Hours
Rik	w12.2022	gross avail.	40

Also consider relation *Factgross(fact)* containing tuple *(”gross availability”)*:

Fact
gross avail.

The query achieving this insertion is:
$DC001_1 \leftarrow DC001_0 \cup (\pi_{employee,fact,week,hours \leftarrow grossavail.}(DC005 \times Factgross))$

2 - Compute net availability per employee per week
Relation to update: *DC001(employee, fact, week, hours)*
Example of relation:

Employee	**Fact**	**Week**	**Hours**
Rik	gross avail.	w12.2022	40
Rik	holiday	w12.2022	12
Rik	education	w12.2022	8

Then the following tuple should be inserted into *DC001*

Employee	**Fact**	**Week**	**Hours**
Rik	net avail.	w12.2022	20

Also consider relation *Factnet(fact)* containing tuple *("net availability")*:

Fact
net avail.

The expression achieving this insertion is:

$$
\begin{aligned}
&DC001_2 \leftarrow DC001_1 \cup \\
&\pi_{a.employee, d.fact, a.week, hours \leftarrow (a.hours - b.hours - c.hours)} \\
&\quad \sigma_{a.fact = grossavail. \wedge b.fact = holiday \wedge c.fact = education} \\
&\qquad (\rho_a(DC001_1) \bowtie_{a.employee = b.employee \wedge a.week = b.week} \\
&\qquad \rho_b(DC001_1) \bowtie_{b.employee = c.employee \wedge b.week = c.week} \\
&\qquad \rho_c(DC001_1) \times \\
&\qquad \rho_d(Factnet))
\end{aligned}
$$

3 - Rollup hours per employee over weeks to year
Relation to update: *DC001(employee, fact, week, hours)*
Example of relation:

Employee	**Fact**	**Week**	**Hours**
Rik	holiday	w12.2022	12
Rik	holiday	w13.2022	4
Rik	gross avail.	w12.2023	8
Anne	net avail.	w12.2022	20
Anne	net avail.	w30.2022	40

Then the following tuple should be inserted into *DC001*

Employee	**Fact**	**Week**	**Hours**
Rik	holiday	2022	16
Rik	gross avail.	2023	8
Anne	net avail.	2022	60

We assume that for a week on the week level the week and year value can be accessed by $week.w$ and $week.y$ respectively.

The insertion can be achieved with expression:
$DC001_3 \leftarrow DC001_2 \cup \pi_{employee, week \leftarrow week.y, fact, hours \leftarrow (\mathcal{G}\mathbf{sum}(hours))}(DC001_2)$

Important to note: the query groups by the $employee$, $week$ and $fact$ dimensions. These are the same as as the projected dimensions, (except for hours). Since the group by dimensions always will follow from the projected dimensions, these are ommited before the group by ($\mathcal{G}$) symbol.

4 - Fill in hours where employee and week correspond to team
Relation to create: *DC003(employee, fact, team, week, hours)*

Input relation: *DC001(employee, fact, week, hours)*
Example of relation:

Employee	Fact	Week	Hours
Rik	holiday	w12.2022	12
Rik	holiday	w13.2022	4
Rik	gross avail.	w12.2023	8
Anne	net avail.	w12.2022	20
Anne	net avail.	w30.2022	40

Input relation: *R001(employee, week, team)*
Example of relation:

Employee	Week	Team
Rik	w12.2022	Dev1
Rik	w13.2022	Dev1
Rik	w12.2023	Dev3
Anne	w12.2022	Con1
Anne	w30.2022	Con2

Then the following relation should be unified with *DC003*:

Employee	Fact	Team	Week	Hours
Rik	holiday	Dev1	w12.2022	12
Rik	holiday	Dev1	w13.2022	4
Rik	gross avail.	Dev3	w12.2023	8
Anne	net avail.	Con1	w12.2022	20
Anne	net avail.	Con2	w30.2022	40

The expression achieving this unification is:
$DC003_1 \leftarrow \pi_{employee,fact,team,week,hours}(DC001_2 \bowtie R001)$

5 - Sum hours over employees to team (per week)
Relation to create: *DC002(fact, team, week, hours)*

Input relation: *DC003(employee, fact, team, week, hours)*
Example of relation:

Employee	Fact	Team	Week	Hours
Rik	holiday	Con1	w12.2022	12
George	holiday	Con1	w12.2022	4
Rik	education	Con2	w13.2022	8
Anne	education	Dev1	w12.2022	20
Susan	gross avail.	Dev1	w12.2022	40

Then the following relation should be unified with *DC002*:

Fact	**Team**	**Week**	**Hours**
holiday	Con1	w12.2022	16
education	Con2	w13.2022	8
education	Dev1	w12.2022	20
gross avail.	Dev1	w12.2022	40

The unification can be achieved with expression:
$DC002_1 \leftarrow \pi_{fact,team,week,hours \leftarrow (\mathcal{G}\mathbf{sum}(hours))}(DC003_1)$

A query representing a sub-model defined by Assemble's standard *Aggregation* operation, can be constructed by following the same structure as the queries of sub-models 4 and 5. Namely, for relations α*(<u>ar1</u>, x)*, β*(<u>ar2</u>, x)*, γ*(<u>y</u>, z)* and aggregation function δ where $ar1$ and $ar2$ are identical sets of dimensions, but for 1 dimension in both sets: $y \in ar1$ and $z \in ar2$ and x is the dimension value to which δ is applied, then β can be computed from α with query:
$\beta \leftarrow \pi_{ar2,x \leftarrow (\mathcal{G}\delta(x))}(\alpha \bowtie \gamma)$

6 - Rollup hours over team to levels department and total
Relation to update: *DC002(<u>fact, team, week</u>, hours)*
Example of relation:

Fact	**Team**	**Week**	**Hours**
education	Con1	w12.2022	16
education	Dev3	w12.2022	8
education	Dev1	w12.2022	20
gross avail.	Dev1	w12.2022	40

Also consider team - department relation $TeamDepartment$:

Team	**Department**
Dev1	Development
Dev2	Development
Dev3	Development
Con1	Consultancy
Con2	Consultancy
Con3	Consultancy
Con4	Consultancy
Mar1	Marketing
Mar2	Marketing

$Teamtotal$

Team
total

Then the following relation should be unified with *DC002*:

Fact	**Team**	**Week**	**Hours**
education	Consultancy	w12.2022	16
education	Development	w12.2022	28
gross avail.	Development	w12.2022	40
gross avail.	Total	w12.2022	40
education	Total	w12.2022	44

The expression achieving this unification is:
$DC002'_2 \leftarrow \pi_{fact,team\leftarrow department,week,hours\leftarrow(\mathcal{G}\mathbf{sum}(hours))}(DC002_1 \bowtie TeamDepartment)$
$DC002_2 \leftarrow DC002_1 \cup DC002'_2 \cup \pi_{fact,b.team,week,hours\leftarrow(\mathcal{G}\mathbf{sum}(hours))}(\rho_a(DC002'_2) \times \rho_b(Teamtotal))$

7 - Compute adjusted hours
Relation to create: *DC007(fact, team, week, adjusted_hours)*

Input relation: *DC002(fact, team, week, hours)*
Example of relation:

Fact	**Team**	**Week**	**Hours**
holiday	Con1	w12.2022	16
education	Dev3	w12.2022	8
education	Dev1	w12.2022	20
gross avail.	Dev1	w12.2022	40
holiday	Consultancy	w12.2022	16
education	Development	w12.2022	28
gross avail.	Development	w12.2022	40

Input relation: *DC006(fact, week, percentage)*
Example of relation:

Fact	**Week**	**Percentage**
holiday	w12.2022	0
education	w12.2022	5
gross avail	w12.2022	12

Then the following relation should be unified with *DC007*:

Fact	**Team**	**Week**	**Adjusted_Hours**
holiday	Con1	w12.2022	16.00
education	Dev3	w12.2022	8.40
education	Dev1	w12.2022	21.00
gross avail.	Dev1	w12.2022	44.80
holiday	Consultancy	w12.2022	16.00
education	Development	w12.2022	29.40
gross avail.	Development	w12.2022	44.80

The expression achieving this unification is:
$DC007_1 \leftarrow \pi_{fact,team,week,adjusted\_hours\leftarrow(1+percentage)/100}(DC002_2 \bowtie DC006)$

8 - Compute moving total
Relation to create: *DC004(fact, team, week, hours)*

Input relation: *DC002(fact, team, week, hours)*
Example of relation:

Fact	**Team**	**Week**	**Hours**
holiday	Dev1	w52.2021	4
holiday	Dev1	w1.2022	12
holiday	Dev1	w2.2022	4
holiday	Dev1	w3.2022	8
holiday	Dev1	w4.2022	20
holiday	Dev1	w5.2022	40
net avail.	Consultancy	w14.2022	10
net avail.	Consultancy	w15.2022	30
net avail.	Consultancy	w16.2022	10
net avail.	Consultancy	w17.2022	24
net avail.	Consultancy	w18.2022	6
net avail.	Consultancy	w19.2022	8

Then the following relation should be unified with *DC004*:

Fact	**Team**	**Week**	**Hours**
holiday	Dev1	w52.2021	4
holiday	Dev1	w1.2022	16
holiday	Dev1	w2.2022	20
holiday	Dev1	w3.2022	28
holiday	Dev1	w4.2022	48
holiday	Dev1	w5.2022	54
net avail.	Consultancy	w14.2022	10
net avail.	Consultancy	w15.2022	40
net avail.	Consultancy	w16.2022	50
net avail.	Consultancy	w17.2022	74
net avail.	Consultancy	w18.2022	80
net avail.	Consultancy	w19.2022	78

Also consider relation *Yearlengths($\underline{year,\ length}$)* with example:

Year	**Length**
2021	52
2022	52

The expression achieving this unification is:
$002withYL \leftarrow \pi_{fact,team,week,hours,length}(DC002_2 \bowtie_{DC002_2.week.y=Yearlengths.year} Yearlengths)$

$\mathcal{P} \leftarrow (a.week - b.week \geq 0 \land a.week - b.week < 5) \lor (a.year = b.year + 1 \land a.week + b.length - b.week \geq 0 \land a.week + b.length - b.week < 5)$
$\gamma_1 \leftarrow \rho_a(DC002) \bowtie_{a.fact=b.fact \land a.team=b.team \land \mathcal{P}} \rho_b(002withYL)$
$DC004_1 \leftarrow \pi_{a.fact,a.team,a.week,hours \leftarrow (\mathcal{G}\mathbf{sum}(hours))}(\gamma_1)$

9 - Rollup hours per team over week to year
Relation to update: *DC002($\underline{fact,\ team,\ week}$, hours)*
Example of relation:

Fact	**Team**	**Week**	**Hours**
holiday	Con1	w12.2022	16
holiday	Con1	w13.2022	8
education	Dev3	w12.2022	21
gross avail.	Dev1	w12.2022	44
holiday	Consultancy	w12.2022	16
education	Development	w12.2022	29
education	Development	w13.2022	10
net avail.	Development	w12.2022	44

Then the following relation should be unified with *DC002*:

Fact	**Team**	**Week**	**Hours**
holiday	Con1	2022	24
education	Dev3	2022	21
gross avail.	Dev1	2022	44
holiday	Consultancy	2022	16
education	Development	2022	39
net avail.	Development	2022	44

The unification can be achieved with expression:
$DC002_3 \leftarrow DC002_2 \cup \pi_{fact,team,week \leftarrow week.y,hours \leftarrow (\mathcal{G}\mathbf{sum}(hours))}(DC002_2)$

10 - Rollup adjusted hours per team over week to year
Relation to update: *DC007($\underline{fact,\ team,\ week}$, adjusted_hours)*
Example of relation:

Fact	**Team**	**Week**	**Adjusted_Hours**
holiday	Con1	w12.2022	16.00
holiday	Con1	w13.2022	8.00
education	Dev3	w12.2022	21.00
gross avail.	Dev1	w12.2022	44.80
holiday	Consultancy	w12.2022	16.00
education	Development	w12.2022	29.40
education	Development	w13.2022	10.20
net avail.	Development	w12.2022	44.80

Then the following relation should be unified with *DC007*:

Fact	**Team**	**Week**	**Adjusted_Hours**
holiday	Con1	2022	24.00
education	Dev3	2022	21.00
gross avail.	Dev1	2022	44.80
holiday	Consultancy	2022	16.00
education	Development	2022	39.60
net avail.	Development	2022	44.80

The unification can be achieved with expression:
$DC007_2 \leftarrow DC007_1 \cup \pi_{fact,team,week \leftarrow week.y,adjusted\_hours \leftarrow (\mathcal{G}\mathbf{sum}(hours))}(DC007_1)$

11 - Compute percentage facts moving total
Relation to create: *DC008($\underline{percentage\_fact,\ team,\ week}$, percentage)*

Input relation: *DC004(fact, team, week, hours)*
Example of relation:

Fact	**Team**	**Week**	**Hours**
holiday	Dev1	w3.2022	28
education	Dev1	w3.2022	0
gross avail.	Dev1	w3.2022	40
holiday	Consultancy	w15.2022	5
gross avail.	Consultancy	w15.2022	40

Then the following relation should be *DC008*:

Percentage_Fact	**Team**	**Week**	**Percentage**
%holiday	Dev1	w3.2022	70.00
%education	Dev1	w3.2022	0.00
%holiday	Consultancy	w15.2022	12.50

Also consider relation *FactPercentagefacts(Fact, Percentage_Fact)* with example:

Fact	**Percentage_Fact**
holiday	%holiday
education	%education

The expression achieving this unification is:
The unification can be achieved with expression:
$004withPF \leftarrow DC004_1 \bowtie FactPercentagefacts$
$\gamma_1 \leftarrow \rho_a(004withPF) \bowtie_{a.team=b.team,a.week=b.week} \rho_b(DC004_1)$
$DC008 \leftarrow \pi_{a.percentage\_fact,a.team,a.week,percentage \leftarrow ((a.hours/b.hours)*100)} \sigma_{b.fact=grossavail.}(\gamma_1)$

12 - Copy team per employee per week
Relation to create: *DC010(employee, team, week, value)*

Input relation: *R001(employee, week, team)*
Example of relation:

Employee	**Week**	**Team**
Rik	w12.2022	Dev1
Rik	w13.2022	Dev1
Rik	w12.2023	Dev3
Anne	w12.2022	Con1
Anne	w30.2022	Con2

Input relation: $Valuetrue$

Value
true

Then the following relation should be unified with *DC010*:

Employee	**Team**	**Week**	**Value**
Rik	Dev1	w12.2022	true
Rik	Dev1	w13.2022	true
Rik	Dev3	w12.2023	true
Anne	Con1	w12.2022	true
Anne	Con2	w30.2022	true

In order to not unnecessarily inflate the size of the data set, instead of having a false value in cells for all employee, team, week combinations that do not coincide, this cell holds an *na* value. This is reflected by the data set above, where tuples with such combinations do simply not exist.

The unification can be achieved with expression:
$DC010 \leftarrow R001 \times Valuetrue$

13 - Count amount of employees per team per week
Relation to create: *DC009(team, week, count)*

Input relation: *DC010(employee, team, week, value)*
Example of relation:

Employee	**Team**	**Week**	**Value**
Rik	Dev1	w12.2022	true
Anne	Dev1	w12.2022	true
Anne	Dev2	w13.2022	true
George	Dev3	w12.2023	true
George	Dev3	w13.2022	true

Then *DC009* should become the following relation:

Team	**Week**	**Count**
Dev1	w12.2022	2
Dev2	w13.2022	1
Dev3	w12.2023	1
Dev3	w13.2022	1

The creation can be achieved with expression:
$DC009 \leftarrow \pi_{team,week,count \leftarrow (\mathcal{G}_{\mathbf{count}(employee)})}(DC010)$

14 - Determine empty teams per week
Relation to create: *DC011(team, week, value)*

Input relation: *DC010(employee, team, week, value)*
Example of relation:

Employee	**Team**	**Week**	**Value**
Rik	Dev1	w12.2022	true
Anne	Con1	w14.2022	true
Anne	Dev2	w13.2022	true
George	Dev2	w12.2022	true
George	Dev2	w13.2022	true

Input relation: *TeamWeekCombos(team, week)* Example of relation:

Team	**Week**
Dev1	w12.2022
Dev1	w13.2022
Dev1	w14.2022
Dev2	w12.2023
Dev2	w13.2022
Dev2	w14.2022
Con1	w12.2022
Con1	w13.2022
Con1	w14.2022

Input relation: $Valuetrue$

Value
true

Then *DC011* should become the following relation:

Team	**Week**	**Value**
Dev1	w13.2022	true
Dev1	w14.2022	true
Dev2	w14.2022	true
Con1	w12.2023	true
Con1	w13.2022	true

The creation can be achieved with expression:
$DC011 \leftarrow (TeamWeekCombos - \pi_{team,week}(DC010)) \times Valuetrue$