COMPUTING CANDIDATE KEYS OF RELATIONAL OPERATORS FOR OPTIMIZING REWRITE-BASED PROVENANCE COMPUTATION - KEY PROPERTY MODULE -

BY

ANDREA CORNUDELLA RAVETLLAT

Master's Thesis in
Telecommunications Engineering in the Graduate College of the
Illinois Institute of Technology and
Universitat Politecnica de Catalunya (ETSETB)

Approved: Professor Dr. Glavic
Advisor

Chicago, Illinois
2015

ACKNOWLEDGMENT

I would like to express my special appreciation and thanks to my advisor Professor Dr. Glavic, who has supported me throughout the entire process. I have learned a lot from him while being guided and advised on every step to do to in every stage of my project, which would have been impossible to accomplish without his help.

I would also like to thank the opportunity given by the UPC to study one semestre abroad, which I really think has been a great opportunity to me from all points of views.

It must be mentioned too the enormous gratitude I have towars my family and friends who have been with me during these months while I was working on my thesis in IIT, bearing me patiently enough, giving me strenghth to strive towards my goal.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

English

Data provenance provides information about the origin of data, and has long attracted the attention of the database community. It has been proven to be essential for a wide range of use cases from debugging of data and queries to probabilistic databases. There exist different techniques for computing the data provenance of a query. However, even sophisticated database optimizers are usually incapable of producing an efficient execution plan for provenance computations because of their inherent complexity and unusual structure. In this work, I develop the key property module, as part of the heuristic optimization techniques for rewrite-based provenance systems to address this problem and present an implementation of this module in the GProM provenance middle-ware system. The key property stores the set of candidate keys for the output relation of a relational algebra operator. This property is important for evaluating the precondition of many heuristic rewrite rules applied by GProM, e.g., rules that reduce the number of duplicate removal operators in a query. To complete this work, I provide an experimental evaluation which confirms that this property is extremly useful for improving the performance at game provenance.

Català

La procedència de dades proporciona informació sobre l'origen de les dades, i ha atret molt l'atenció de la comunitat de recerca en bases de dades. S'ha demostrat que és essencial per a una àmplia gamma de casos, des de debugging de dades i consultes fins a bases de dades probabilístiques. Existeixen diferents tècniques per al càlcul de la procedència de dades d'una consulta. No obstant això, fins i tot els optimitzadors de bases de dades sofisticats solen ser incapaços de produir un pla d'execució eficient per a càlculs de procedència a causa de la seva complexitat inherent i la seva estructura inusual. Al llarg d'aquest treball, desenvolupo un mòdul per inferir la propietat clau als operadors, com a part de les tècniques d'optimització heurística per a sistemes de procedència de dades basades en la reescriptura per fer front al problema d'optimització i presentar una implementació d'aquest mòdul en el sistema middleware de procedència GProM. La propietat clau emmagatzema el conjunt de claus candidates per a la relació de sortida d'un operador d'àlgebra relacional. Aquesta propietat és important per avaluar la condició prèvia de moltes regles de reescriptura heurístiques aplicats pel sistema GProM, per exemple, les normes que redueixen el nombre d'operadors d'eliminació de duplicats en una consulta. Per completar aquest projecte, proporciono una avaluació experimental que confirma que aquesta propietat és extremadament útil per millorar el rendiment en el joc de procedència.

Español

La procedencia de datos proporciona información sobre el origen de los datos, y ha atraído mucho la atención de la comunidad de investigación en bases de datos. Se ha demostrado que es esencial para una amplia gama de casos, desde debugging de datos y consultas hasta bases de datos probabilísticos. Existen diferentes técnicas para el cálculo de la procedencia de datos de una consulta. Sin embargo, incluso los optimizadores de bases de datos sofisticados suelen ser incapaces de producir un plan de ejecución eficiente para cálculos de procedencia debido a su complejidad inherente y su estructura inusual. A lo largo de este trabajo, desarrollo el módulo para inferir la propiedad clave a los operadores, como parte de las técnicas de optimización heurística para sistemas de procedencia de datos basados en la reescritura para hacer frente al problema de optimización y presentar una implementación de este módulo en el sistema middleware de procedencia GProM. La propiedad clave almacena el conjunto de claves candidatas para la relación de salida de un operador de álgebra relacional. Esta propiedad es importante para evaluar la condición previa de muchas reglas de reescritura heurísticas aplicados por el sistema GProM, por ejemplo, las normas que reducen el número de operadores de eliminación de duplicados en una consulta. Para completar este trabajo, proporciono una evaluación experimental que confirma que esta propiedad es extremadamente útil para mejorar el rendimiento en el juego de procedencia.

CHAPTER 1
INTRODUCTION

A large portion of data generated and stored by scientific databases, data warehouses, and workflow management systems is not entered manually by a user, but is derived from existing data using complex transformations. Understanding the semantics of such data and estimating its quality is not possible without extensive knowledge about the data's origin and the transformations that were used to create it. In general, for every application domain where data is heavily transformed, data provenance is of essential importance. Data provenance is information that describes how a given data item was produced. The provenance includes source and intermediate data as well as the transformations involved in producing the concrete data item. It can be used to estimate the quality of data, determine trust measures of data, to gain additional insights about it, or to trace errors in transformed data back to its origins. A complete record of provenance in scientific computations can help determine the quality and the trust one places on the scientific result, typically regarded to be as important as the result itself. Recording provenance information for query results, that explains the computational process leading to their generation, is now a common technique.

For example, consider a relation storing employee salaries. The relation is subject to complex transactional updates such as calculating tax, applying tax deductions, multiplying rates with working hours, and so on. How do we know that the information in the current version of the relation is correct? If one employee salary is wrong, how do we know which update(s) or data caused the error? Data provenance, by providing a full record of the derivation history of data, makes it possible to track the cause of the error.

The standard for database provenance is to model provenance as annotations on data and compute the provenance for the outputs of an operation by propagating those annotations. Many provenance systems use a relational encoding of provenance annotations. These systems apply query rewrite techniques to transform a regular query into a query that propagates input annotations to produce the result of q annotated with provenance.

Provenance rewrites and other aforementioned techniques generate queries with unusual access patterns and operator sequences. Even sophisticated database optimizers are not capable of producing reasonable plans for such queries. For example, after provenance rewriting, the generated query expression may contain a large number of window operations interleaved with joins. Regular database queries written by users or automatically generated by tools (e.g. reporting tools) do not usually generate this kind of patterns. This is the reason why most database optimizers are incapable of simplifying such queries and will not explore relevant parts of the plan space. Thus, while provenance rewrites enable easy implementation of provenance support for databases without the need of modifying the database system itself, their performance is often far from optimal.

In this project we are addressing this problem through the development of novel heuristic and cost-based optimization techniques and their implementation in the GProM system. My personal contribution to this project is focused on a specific module for the heuristic rules, using the cost-based optimizer to demonstrate its effectiveness. In the following sections I give an overview of the system and then discuss the need for heuristic and cost-based optimizations based on two examples. I also give an insight on how we will approach this need.

1.1 The GProM System

GProM (**G**eneric **Pro**venance **M**iddleware) is a middleware system that enables computation of provenance for queries, updates, and transactions over several database back-ends (e.g., Oracle). GProM is the first system capable of computing the prove-

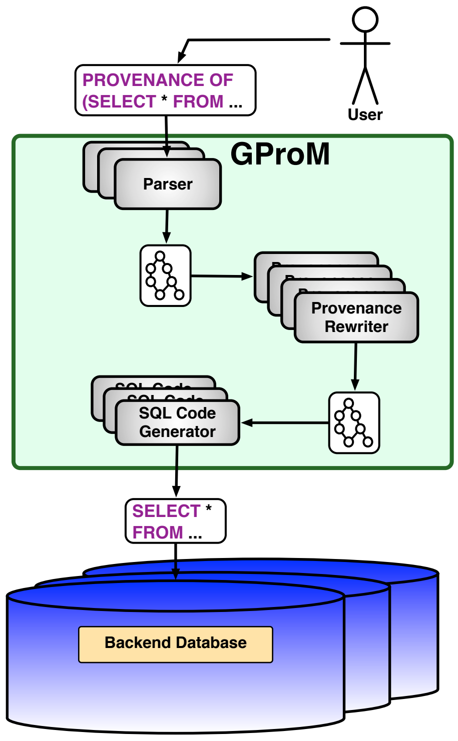


Figure 1.1: GProM Architecture

nance of concurrent database transactions. It uses annotation propagation and query rewrite techniques for computing, querying, storing, and translating the provenance of SQL queries, updates, transactions, and across transactions. It is database independent and supports multiple types of provenance computation. It uses a query rewrite technique which rewrites the algebra tree and then translates it into SQL code. Internally all queries are represented as relational algebra graphs in GProM. Figure 1.1 shows the architecture of the system. User queries (including requests for provenance) are parsed and transformed into relational algebra graphs. We refer to this representation as *AGM* (Algebra Graph Model). The provenance rewrite techniques that take the input request and rewrite it to propagate provenance, all operate on the AGM representation of a query. The heuristic and cost-based optimizations also operate on AGM graphs. After a query is optimized by our optimizer, the AGM graph is translated into executable SQL code in the dialect of the back-end database system. The ultimate goal of this project is to develop an optimizer module that can be used to improve the performance of different types of provenance computations. GProM is an ideal platform for this type of research, because it already supports several different provenance types (e.g., provenance for queries using the semiring model [1], the first implementation of provenance for transactions, and game provenance [2]) and back-end languages (currently Oracle's SQL dialect and datalog). Since database systems already employ quite evolved optimization techniques, it is essential that the proposed optimizations do not hinder the database optimizer in exploring its plan search space. In the following, I provide examples to help the reader understand some of the provenance types supported by GProM.

Example 1. *Table 1.1 shows an example database with relations* `shop`*,* `sales`*, and* `items`*. We run the following query expressed in relational algebra which computes the total sales per shop by joining the shop, sales, and items relations and aggregating the price of sold items grouped by shop name.*

$$\begin{aligned} q &= \Pi_{name,sum(price)}(match) \\ match &= \sigma_{names=sName \wedge itemId=id}(prod) \\ prod &= shop \times sales \times items \end{aligned} \tag{1.1}$$

The output of this query (also shown in Table 1.1) contains two tuples. In the first tuple, the sum(price) is 120, in the second tuple, the sum(price) is 50. The user may decide to request the provenance of these tuples to understand from which inputs they were derived.

Using, e.g., GProm, we get provenance as shown in Table 1.2. Here each result tuple of the query is paired with tuples from the provenance. Attributes from the input

shop

name	numEmpl
Merdiers	3
Joba	14

Sales

sName	itemId
Merdiers	1
Merdiers	2
Merdiers	2
Joba	3
Joba	3

Items

id	price
1	100
2	10
3	25

result q

name	sum(price)
Merdiers	120
Joba	50

Table 1.1: Example database

relations have been renamed to indicate that they store provenance (here represented by $P(name)$). From the first three rows we can see that the sum 120 *was computed by adding the prices of three input tuple combinations* $(100 + 10 + 10)$*. Note that the original result tuple has been duplicated to fit in all its provenance. Similarly, for the second result tuple (the last two tuples in the provenance) the sum* 50 *was computed based on two tuples from the input* $(50 = 25 + 25)$*. Also we can see which input tuples have been joined together by the query before the aggregation.*

As mentioned before, GProM also supports provenance computation for transactions. If a user wants to request the provenance for a transaction T, the transaction reenactor of GProM extracts the list of SQL statements executed by T from the audit log of the backend database and constructs a reenactment query $q(T)$ that simulates the effects of these statements. We use the provenance rewriter to rewrite $q(T)$ into a query $q(T)+$ that computes the provenance of the reenacted transaction.

1.2 The Need of Optimization

We motivate the need for heuristic optimizations by means of a simplified real world example we encountered in GProM. This example illustrates the dire need for applying optimization, because the unoptimized query is known to have an execution time

result		**prov. shop**		**prov. sales**		**prov. items**	
name	sum(price)	P(Name)	P(NumEmpl)	P(SName)	P(ItemId)	P(Id)	P(Price)
Merdies	120	Merdiers	3	Merdies	1	1	100
Merdies	120	Merdiers	3	Merdies	2	2	10
Merdies	120	Merdiers	3	Merdies	2	2	10
Joba	50	Joba	14	Joba	3	3	25
Joba	50	Joba	14	Joba	3	3	25

Table 1.2: Provenance result of example database

```
UPDATE R SET A=A−5 WHERE B=2;
UPDATE R SET A=A+1 WHERE B=1;
COMMIT;
```

(a) Example Transaction T_1

$$\begin{array}{l} \Pi_{if(B=1)\ then\ A+1\ else\ A \rightarrow A,B} \\ | \\ \Pi_{if(B=2)\ then\ A-5\ else\ A \rightarrow A,B} \\ | \\ R \end{array}$$

(b) Simplified Provenance Computation for T_1

Before T_1

A	B
2	1
3	2
4	2

After T_1

A	B
3	1
-2	2
-1	2

(c) Relation R before and after Transaction T_1

Figure 1.2: Example Transaction T_1

in most databases (actually all database systems we tested including Oracle and PostgreSQL) that is larger than the age of the universe while the optimized query runs in milliseconds.

Example 2 (Heuristic). *Figure 1.2(a) shows an SQL transaction consisting of two updates followed by a commit (that instructs the database to persist the changes made by the transaction). Figure 1.2(c) shows the relation R before and after executing transaction T_1. If asked to compute the provenance for this transaction, the reenactment module of GProM will generate the algebra expression that is simplified in Figure 2(b). Note that the conditional expressions would be expressed using CASE in*

SQL. Explaining the details of provenance computation for transactions is far beyond the scope of this paper. Thus, we simplified this expression to only show the part relevant to understand the problem and will only explain basic concepts are necessary. This expression seems simple enough and a reader with database background may be mislead to expect that it would run in linear time in the size of relation R. However, most database optimizers will try to merge the two projections into one by substituting the references to attributes in the projection expressions of the top-most projection with their definitions in the lower projections. For instance, this would replace every reference to A in $if(B = 1)\ then\ A + 1\ else\ A$ *with* $if(B = 2)\ then\ A - 5\ else\ A$ *leading to the expression:*

$$\begin{aligned}
&if(B = 1)\\
&\ then\ (if(B = 2)\ then\ A - 5\ else\ A) + 1\\
&\ else\ (if(B = 2)\ then\ A - 5\ else\ A)
\end{aligned}$$

Note that in the resulting expression the attribute A from relation R is referenced 4 times. While for two levels this is not really a problem, consider what would happen if we had n *levels instead of* 2*. Then the resulting expression would be exponential in* n*, because every merge step doubles the number of references to A. In practice, we saw this in the first prototype implementation of GProM, when trying to execute provenance computations for transactions which have this pattern. The result was that the optimizer of Oracle never finished generating a plan for the query (we confirmed the same behavior for PostgreSQL). This is quite a surprising result for a query which has inherently linear complexity in* n *and* $|R|$*. Essentially, Oracle failed to implement a safety check when applying merging projections. One heuristic rule* *that has been implemented in GProM is factoring of attribute references to reduce the total number of references to attributes in projection expressions. Applied to, e.g.,* $if(B = 1)$ *then* $A + 1$ *else* A *we can factor the common reference to* A *in the then and else branch to get* $A + (if(B = 1)$ *then* $+1$ *else* $0)$. *After attribute factoring, the projections can safely be merged. Furthermore, we determine when merging is unsafe and force the database to materialize intermediate results in this case.*

Heuristic rules are a great tool for simplifying query expressions as well as rewriting them into a form that is easy to understand and optimize by standard database optimizers. However, this approach is not applicable if we have to choose between alternative ways of expressing a provenance computation and none of these choices is clearly superior. Ideally, we want to be able to make an informed choice based on which choice has the lower expected cost for the query at hand. This is when the cost-based optimizer comes in.

Example 3 (Cost-based)**.** *GProM implements two different ways of rewriting aggregation in queries for provenance computation - one is based on joining and the other one uses window functions. There is no clear winner among these two methods, it really depends on the size of the input database, value distributions of attributes, and the structure of the query. Using heuristic rules will make no sense, because half of time we would make an inferior choice. A cost-based optimizer, however, can determine which method is better for an input query. Theoretically, a database optimizer could be able to determine that these two ways of expressing the provenance computation are equivalent. However, in practice no optimizer considers such equivalences. As we will demonstrate in the experimental evaluation, the cost of these two ways of rewriting aggregation are significantly different (sometimes many orders of magnitude) and our cost-based optimizer is capable of making the right choice.*

1.3 Our Approach

The heuristic optimization techniques devised in this work consist of simple equivalence rules some of which require inference of certain properties for operators in a query to test whether they are applicable. While some of these rules are novel (or at least atypical), quite some of these rules are based on standard equivalences. The reason for implementing these standard rules (which may be applied in similar form by the database optimizer) is that it may be necessary to apply a standard rule to open up opportunities for applying our optimizer's novel rules.

Our solution allows our cost-based optimizer to generate alternative SQL queries and to use the database back-end's optimizer to generate the best plan for each query and give us a cost estimate. We then execute the query with the lowest estimated cost. The only drawback of this approach is that the overhead we pay per generated query is to high to allow for an exploration of a large plan space. We address this problem by 1) only consider choices where the choice is likely to significantly affect runtime and 2) by stopping optimization when the ratio between time spend on optimization and expected run time of the current best plan becomes to large. Since, we wanted to integrate cost-based choices with existing rewrite code (and future rewrite methods) one major goal in developing the cost-based optimizer was to minimize changes to the existing code. Our optimizer runs independently from the rewrite code and only requires changes of a few lines of code to register a new choice.

The remainder of this paper is organized as follows. I present background and notation in Section 2. Related work on data provenance and query optimization is covered in Section 3. The heuristic optimization techniques are presented in Section 4, making special emphasis on the Key Property inference. I discuss an experimental evaluation of my work in Section 5. Finally I conclude in Section 6, with conclusions and future work.

CHAPTER 2
BACKGROUND

To better understand the research topic, readers are expected to have knowledge including but not limited to database systems, relational algebra and optimization techniques. Some key terminologiy will be explained in this chapter and examples will be provided.

In this chapter we will introduce the Relational Data Model and Relational Algebra, which are essential topics to fully understand the research developed in this thesis. We also introduce general Query Optimization concepts, which will be the basis of the thesis.

2.1 Relational Data Model

The storage and management of information is one of the most important issues that has been approached throughout the last decades. The manner in which information is organized can have a profound effect on how easy it is to access and manage. Perhaps the simplest but most versatile way to organize information is to store it in tables. The relational model is centered on this idea: the organization of data into collections of two-dimensional tables called relations.

The relational data model is widely used around the world for data storage and processing. This model is simple and it has all the properties and capabilities required to process data with storage efficiency. That is the reason why this model is used in most commercial and open-source database systems.

First proposed by E.F. Codd [3] in 1969, it is a method of structuring data using relations, which are sets of tuples, making an analogy to grid-like mathematical structures consisting of columns and rows. Codd proposed the relational model for IBM, but he had no idea how extremely vital and influential his work would become

Student

s_id	s_fname	s_lastname	s_gpa
1221	John	Smith	3
1222	Robert	Johnson	3.8
1223	Alice	Williams	3.5

Department

d_id	d_name
2	Mathematics
3	Physics

Course

c_id	c_name	c_credits	department_id
551	Calculus	3	2
552	Algebra	4	2

Enrollment

student_id	course_id
1221	551
1222	551
1223	552

Table 2.1: University Database

as the basis of relational databases.

As noted before, in the relational model, all data must be stored in *relations* (tables). Relations are used to group information about a particular type of entity from a domain. Table 2.1 gives a simple example of a university database which has several relations such as `student`, `course`, `enrollment` and `department`. Each relation has its *schema* (structure) and *instance* (data). The *schema* is simply the list of *attributes* (columns) in the relation i.e., in the `student` relation mentioned above the schema would be `s_id, s_fname, s_lname and s_gpa`. The *instance* is the data that populates the relation, organized into *tuples* (rows). A tuple contains all the data of a single instance of the entity represented by this relation i.e., information about a particular student (1221, John, Smith, 3).

Another major characteristic of the relational model is the usage of *keys*. These are specially designated columns within a relation, used to order data or relate data to other relations. One of the most important keys is the *primary key*, $PK(R)$, which is used to uniquely identify each row within a relation. In the `student` relation the `s_id` would be the primary key ($PK(student) =$ `s_id`).

Besides defining how the data are to be structured as discussed above, the relational model also lays down a set of rules to enforce data integrity, known as integrity constraints. It also defines how the data are to be manipulated (relational calculus or relational algebra).

In the following paragraphs we formally introduce Relational Schema and Instance for better understanding the terminology used in this thesis.

Definition 1 (Relational Schema). *A relation schema $R = (A_1, \ldots, A_n)$ consists of a name (R) and a list of attribute names (A_1 to A_n). The arity of a relation schema is the number of attributes in the schema. A database schema $S = \{R_1, \ldots, R_n\}$ is a set of relation schemas R_1 to R_n.*

Definition 2 (Relational Instance - Set Semantics). *Let $\mathcal{U}$ be a universal domain of values. An instance R of a relation schema SCH(R) (sometimes also called a relation) is a subset of $\mathcal{U}^n$, i.e., a set of tuples with same arity as the schema and values from $\mathcal{U}$. An instance D of a database schema SCH(D) is a set of relation instances - one for each relation schema in SCH(D).*

This definition of relation instance is often called *set semantics*, because each relation is a set of tuples. Implementations of the relational model which use the SQL query language (essentially all implementations of relational databases) use a slightly different model called *bag semantics* where a relation may contain multiple duplicates of the same tuple. Formally, this can be achieved by modeling a relation as a function from $\mathcal{U}^n \to \mathbb{N}$ that associates each tuple with a multiplicity (the number of times it occurs in a relation) and maps tuples that do not occur in the relation to 0.

Definition 3 (Relational Instance - Bag Semantics). *Let $\mathcal{U}$ be a universal domain of values. An instance R of a relation schema SCH(R) under bag semantics is a function $\mathcal{U}^n \to \mathbb{N}$ with finite support $|\{t | R(t) \neq 0\}|$. We use t^m to denote that tuple t occurs with multiplicity m, i.e., $R(t) = m$.*

We add the formal definition of key, as it will be useful to demonstrate some properties later.

Definition 4 (Key)**.** *A set of attributes* $K \subseteq SCH(R)$ *is a key for relation* R *iff* $\forall t, t' \in R : t.k = t'.k \rightarrow t = t'$ *and* $\forall t^n \in \pi_k(R) \rightarrow n = 1$

2.2 Relational Algebra

Relational database systems are expected to be equipped with a query language that can assist its users to query the database instances. Two important relational query languages are relational algebra and relational calculus. Relational algebra is a procedural query language. As the name suggests it is an algebra of relations. There are well-known methods for translating between relational algebra and SQL, the query language used by most database systems. Thus, we can study optimization and provenance computation for relational algebra and the results are guaranteed to translate to SQL.

Relational algebra consists of operators that take instances of relations as inputs and yields instances of relations as outputs. These operators can be combined allowing us to perform complex queries. The fundamental operators we use to represent such queries are shown in the Table 2.2 and discussed in the following lines. An operator can be either unary or binary depending on whether they have one or two inputs respectively.

Selection σ returns a relation containing all and only the tuples of R that fulfills the condition θ. Projection π projects all input tuples on a list of projection expressions. Here, A denotes a list of expressions with potential renaming (denoted by $e \rightarrow a$) and $t.A$ denotes applying these expressions to a tuple t. The union of relations R and S, denoted by $R \cup S$, returns the set of tuples that are in R, in S or in both. The intersection of relations R and S, denoted by $R \cap S$, returns the set

Operator	Definition
σ	$\sigma_\theta(R) = \{t^n \mid t^n \in R \wedge t \models \theta\}$
π	$\pi_A(R) = \{(t.A)^n \mid t^n \in R\}$
$\cup$	$R \cup S = \{(t,s)^{n+m} \mid t^n \in R \wedge s^m \in S\}$
$\cap$	$R \cap S = \{(t,s)^{min(n,m)} \mid t^n \in R \wedge s^m \in S\}$
$-$	$R - S = \{(t,s)^{max(n-m,0)} \mid t^n \in R \wedge s^m \in S\}$
$\times$	$R \times S = \{(t,s)^{n*m} \mid t^n \in R \wedge s^m \in S\}$
γ	${}_G\gamma_A(R) = \{(t.G, agg(G(t)))' \mid t^n \in R\}$ $G(t) = \{s^n \mid s^n \in R \wedge t'.G = t.G\}$
δ	$\delta(R) = \{t^1 \mid t^n \in R \ \wedge n \neq 0\}$
$\bowtie$	$R \bowtie_\theta S \equiv \sigma_\theta(R \times S)$
ω	${}_G\omega_A(R) = \{(t.G, A(G(t)))^n \mid t^n \in R\}$ $G(t) = \{s^n \mid s^n \in R \wedge t'.G = t.G\}$
$\{t\}$	$\{t\} = \{t'\}$

Table 2.2: Relational Algebra Operators

of tuples which are in both relations. Difference $R - S$ returns the set of tuples that are in R and are not in S. Crossproduct $R \times S$ returns all possible combinations of two tuples (one from each relation). Aggregation ${}_G\gamma_A(R)$ groups tuples according to their values in attributes G and computes aggregation function A over each group. Duplicate removal $\delta(R)$, as the name suggests, removes duplicates. A join $R \bowtie_\theta S$ can be equivalently expressed as $\sigma_\theta(R \times S)$, and it returns all combinations of tuples from R and S that match the condition θ. The window operator ${}_G\omega_A(R)$ performs a calculation across a set of tuples that are related to the current tuple. A constant relation operator $\{t\}$ simply returns a new relation with one tuple (the one we have selected).

In the case we wanted to go back to the set semantics, we would use a duplicate removal operator (δ).

The operators union, intersection and difference can only be used with relations that are *union-compatible*. Two relations are union-compatible if (1) they

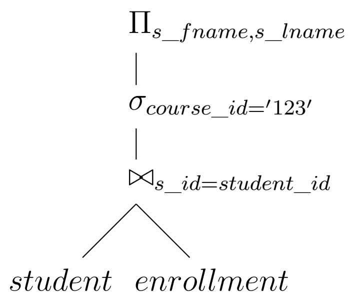

Π_{s_fname,s_lname}
$\sigma_{course\_id='123'}$
$\bowtie_{s\_id=student\_id}$
$student$
$enrollment$

Figure 2.1: Example Algebra Tree

s_fname	s_lname
John	Smith
Robert	Johnson

Figure 2.2: Example Query Result

have the same arity and (2) the corresponding attributes have the same domains. In the previous section we are assuming we just have one domain, but in general this condition would be necessary for SQL.

Example 4 (Relational Algebra)**.** *The relational algebra expression shown below returns the first and last name of the students enrolled in the course with ID 551. We have split the query into two parts: the first part* (j) *matches the students to the enrolled courses, using a join operator. The second part uses selection to narrow down the result to the students enrolled in the course with ID 551 and uses projection to return the attributes we are interested in. The algebra tree representation of this query is shown in Figure 2.1. The result of evaluating this query over the instance from Table 2.1 is shown in Figure 2.2.*

$$\begin{gathered} j = student \bowtie_{s\_id=student\_id} enrollment \\ q_{ex} = \pi_{s\_fname,s\_lname}(\sigma_{course\_id='123'}(j)) \end{gathered} \tag{2.1}$$

2.3 Query Optimization

Given a query, there are many access plans that a database management system can follow to process it and produce its result. All plans are equivalent in terms of their final output but vary in their cost, that is, the amount of time that they need to run. This cost difference can be several orders of magnitude large. Thus, all DBMSs have a module that examines alternative plans and chooses the plan that needs the least amount of time. This module is called the query optimizer.

The same relational algebraic expression can be written in many different ways. When any query is submitted to a relational database system, its query optimizer tries to find the most efficient equivalent expression before evaluating it. *Equivalence* for relational algebra expressions is formally defined below.

Definition 5 (Query Equivalence)**.** *Two relational algebra queries q_1 and q_2 over a database schema $\mathbf{D}$ are called equivalent (denoted by $q_1 \equiv q_2$) if they return the same result over all possible instances D of $\mathbf{D}$:*

$$q_1 \equiv q_2 \Leftrightarrow \forall D : q_1(D) = q_2(D)$$

Note that queries which are equivalent under set semantics are not necessarily equivalent under bag semantics. Unless explicitly stated we assume bag semantics in this work. For cases where we need to distinguish between bag and set equivalence we use a subscript to denote the type, e.g., $q_1 \equiv_{bag} q_2$. This thesis is based on Heuristic Optimization, using rules to transform and rewrite the queries to equivalent ones which may have better performance.

A relational algebra operator can be implemented in many different ways which have different performance characteristics (runtime complexity, number of I/O operations, memory consumption). These different implementations of relational algebra operators are often called *physical operators*. It is well-known that the runtime can vary several orders of magnitude between alternative implementations of a query. Hence, virtually all implementations of relational database systems optimize an input query with the goal to find an efficient plan.

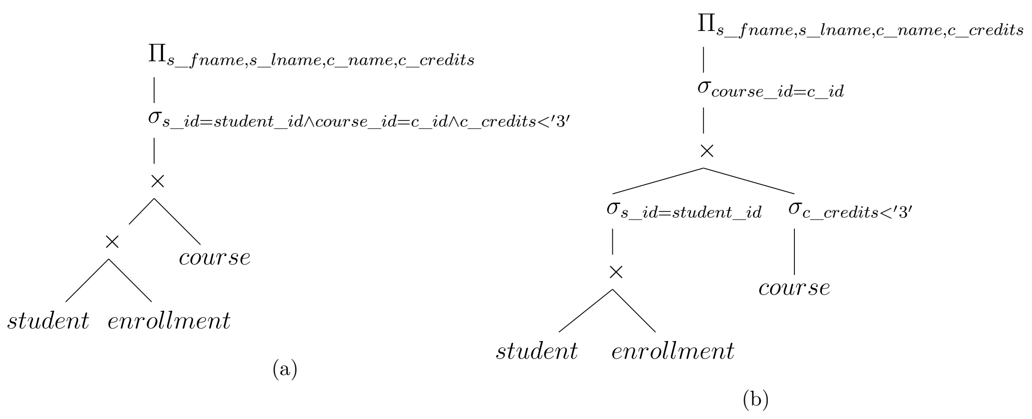

$\Pi_{s\_fname,s\_lname,c\_name,c\_credits}$
$\sigma_{s\_id=student\_id \wedge course\_id=c\_id \wedge c\_credits<'3'}$
$\times$
$\times$
course
student
enrollment
(a)
$\Pi_{s\_fname,s\_lname,c\_name,c\_credits}$
$\sigma_{course\_id=c\_id}$
$\times$
$\sigma_{s\_id=student\_id}$
$\sigma_{c\_credits<'3'}$
$\times$
course
student
enrollment
(b)

Figure 2.3: Example of Heuristic Optimization

Example 5. *Consider the following algebra expression over the database from Table 2.1:*

$$q_1 = student \times enrollment \times course$$
$$q_2 = \sigma_{s\_id=student\_id \wedge course\_id=c\_id \wedge c\_credits<'3'}(q_1) \tag{2.2}$$
$$q = \Pi_{s\_fname,s\_lname,c\_name,c\_credits}(q_2)$$

We illustrate how to transform it into an equivalent algebra expression, Figure 2.3 shows the transformation process. Figure 2.3 (a) is the original algebra tree, after the equivalence rewrite of (a) based on a heuristic rule, (a) is transformed to (b) which is equivalent to (a) but expected to have less cost. The underline idea is that the two cross product operations require lots of space and time to compute which results in a large cost for expression (a). A typical rule for heuristic query optimization is to perform selection and projection operations as early possible before evaluating any joins. The rationale is that implementations of join have super-linear runtime whereas selections have at most linear runtime. Thus, it is beneficial to reduce the number of tuples before joins in order to reduce its cost.

Query optimization is a hard problem for the following reasons: 1) the potential search space is infinite, because there are infinitely many queries that are equivalent to a query and 2) the cost of a query cannot be predicted precisely. The first problem is usually addressed by limiting the search space to equivalences that are deemed most beneficial and the second problem is addressed by using statistics about the distribution of attribute values in the current database instance to efficiently estimate the cost of a plan. Typical optimizers use a combination of heuristic rules and exhaustive cost-based search for particular equivalences. In Section 3.2 we discuss these two types of optimizations in more detail.

CHAPTER 3
RELATED WORK

3.1 Data Provenance

Data provenance is essential in applications such as scientific computing, curated databases, and data warehouses. For this reason tracking the provenance of data has recently attracted the attention of the database community, leading to the development of several systems that provide data provenance functionality for the relational model with different approaches on how to model it.

Data provenance, which represents dependencies between a query's input and output data, is categorized based on the type of dependency that is modeled. Why-provenance models which input tuples are used to create an output tuple. As related work on this type of provenance we need to mention the pioneer Why-provenance model by Buneman et al. [4], Lineage proposed by Cui et al.[5] and the provenance semantics supported by Perm [6]. Where-provenance models where values in an output tuple were copied from. This models were introduced by Buneman et al. [4] and Perm [6]. How-provenance extends Why-provenance with information about how input tuples are used to create an output tuple. The model of the Trio [7] prototype can be classified as How-provenance.

Most of the developed systems implement only one type of provenance. DB-Notes [8] is an annotation management system that uses Where-provenance to propagate annotations. Trio [7] supports uncertainty and provenance. It uses boolean formulas over tuple variables as provenance. The WHIPS data-warehouse prototype [5] implements lineage.

Orchestra [9] is an update-exchange system that uses provenance polynomials to record the provenance of updates exchanged between peers. Orchestra also supports Why-provenance and the model of Trio, because these provenance types can be extracted from provenance polynomials. Perm [6] supports a representative set of provenance semantics including the relational adaptation of Where-provenance, provenance polynomials, and new types of Why, Where, and How [10]. In contrast to Orchestra, generation of these provenance types is supported natively instead of deriving them from a more expressive provenance model. This enables the use of type-specific optimizations during provenance generation for a more efficient execution.

Finally, I will discuss the support that these models can provide for large databases, and how optimization is (or could be) used. In DBNotes [8] provenance annotations for relations in a database are materialized. The method they use allows the system to rely on a DBMS to optimize the execution. However, the SQL query results have to be post-processed to transform them into DBNotes's data model which introduces a potential performance bottleneck. Orchestra [9] uses several SQL queries to implement a single ProQL query and full materialization of provenance information, which limits the scalability of the approach. Trio [7] generates provenance eagerly during query execution. The system materializes the results of each query and creates a separate relation to store its provenance as a mapping between input and output identifiers. WHIPS [5] implements provenance generation as stored procedures that split a query into subexpressions and execute one or more SQL queries to retrieve the lineage of each segment. This separation into multiple queries limits the space of possible optimizations that the underlying DBMS can apply.

Provenance polynomials generated by query evaluation in relational databases have a regular structure that can be exploited for a more succinct representation via algebraic factorizations. On factorization of provenance polynomials [11] is centered in the investigation and use of such factorized representations of provenance polynomials, highlighting key properties and potential benefits of factorized provenance polynomials.

The paper Provenance Minimization [12] studies the core of provenance information, namely the part of provenance that appears in the computation of every query equivalent to the given one. They propose using this compact representation to alleviate practical challenges that arise in data management tools due to the size of provenance information. This paper [12] provides algorithms for such direct computation of the core provenance, improving the efficiency of provenance-based analysis tools, in the sense that they may be fed with smaller provenance polynomials.

Chapman et al. adopts a different angle to improve the efficiency of provenance computation, as shown in Efficient Provenance Storage [13]. They identify important properties of provenance that can be used to considerably reduce the amount of storage required, including the following techniques: a family of factorization processes and two methods based on inheritance, to decrease the amount of storage required for provenance.

Perm [6] supports simple SQL language extensions to let a user specify when (and what) provenance to compute. In Perm, a query over provenance information would usually include a sub-query that generates the provenance. Thus, provenance generation and querying are entangled within a single SQL-PLE query that is rewritten by the system into a single SQL query. This approach allows us to take full advantage of the optimizer of the underlying DBMS. For SQL queries without nesting, it has been demonstrated that the optimizer can significantly improve the performance of provenance queries by, e.g., pushing selections over provenance data into the provenance generation.

For nested sub-queries, we present a set of novel un-nesting and de-correlation optimizations tailored for provenance generation.

These state-of-the-art methods have several shortcomings, including no comprehensive implementation of provenance for relational updates and transactions, no support for automatically determining when and how to compute and store provenance, and supporting only one or a few types of provenance. The GProM system [14] previously introduced, overcomes some of these disadvantages by using annotation propagation and query rewrite techniques for computing the provenance of SQL queries, updates and transactions. It is the first system that supports provenance computation for concurrent transactions, it supports several database back-ends and can be easily extended with new rewrite rules. However it faces an important issue that is faced by many other rewrite based provenance systems: the queries it produces are very hard to optimize by database systems leading to poor performance.

Throughout this thesis we approach this matter by developing optimization techniques targeted at query-rewrite based provenance computations and implement these techniques in GProM.

3.2 Query Optimization

One major criticism of many early DBMSs has been their lack of efficiency in handling the powerful operations they offer, particularly the content-based access to data by queries. Query optimization tries to solve this problem by integrating a large number of techniques and strategies, ranging from logical transformations of queries to the optimization of access paths and the storage of data on the file system level.

As introduced before, optimizers usually combine two types of optimizations: Heuristic and Cost-based optimization. *Heuristic optimization* aims to lower the cost of a query by heuristically applying equivalence-preserving transformations to an input algebra expression. Potentially, the result of such a rewrite is not the best of all the actual solutions, but it is still a valuable one. Using the heuristic rules, the algebra tree can be transformed to an equivalent one which has lower execution cost. *Cost-based optimization* is a widely applied method to optimize the query based on comparing multiple execution plans according to an estimated cost metric. A cost-based query optimizer takes the algebra tree of the input query and produces a set of equivalent plans (physical operator trees which represent equivalent algebra trees), estimates the cost of each plan, and returns the plan with the lowest estimated cost. Even when only a limited set of equivalences is considered, there can be a large number of possible equivalent plans and the throughput or response times for the execution of these plans may vary a lot. Therefore, a judicious choice of an execution by the optimizer is of critical importance. Desirable properties for a cost-based optimizer are: (1) the explored search space is likely to include plans that have low cost, (2) the cost estimation technique is accurate, and (3) the enumeration algorithm is efficient.

The survey written by Jarke, M. and Koch, J. on Query optimization in database systems [15] gives an overview of logical transformation techniques and physical evaluation methods for database queries, using the framework of the relational calculus. This survey shows that a large body of knowledge has been developed to solve the problem of processing queries efficiently in conventional centralized and distributed database systems. However, query optimization research is still an active field. Since relational databases are widely used in a variety of commercial and scientific applications the effectiveness of a query optimizer is of utmost importance to the performance of most enterprises.

The first implementation on query optimization in relational database systems was in the system R [16] prototype developed by IBM. Astrahan et al. [17] and Kooi et al.[18] introduced cost models and selectivity estimation. The first of them describes how System R chooses access paths for both simple (single relation) and complex queries (such as joins), given a user specification of desired data as a boolean expression of predicates.

The System R optimizer [16] used simple statistics, such as minimum and maximum values in a given column, to estimate selectivity factors. Using such simple statistics will produce good selectivity estimates only if the attribute values are uniformly distributed. Since attribute values can have other distributions, it has become commonplace for relational query optimizers to use histograms for estimating selectivity factors. These histograms traditionally have the same width, i.e., [19]. Equi-width histograms also produce erroneous selectivity estimates if the attribute values are not uniformly distributed. This paper [20] shows that even in the case of queries involving multiple attributes, equi-depth histograms are superior to equi-width histograms. This [21] provides a taxonomy of histograms that captures all previously proposed histogram types and indicates many new possibilities (introduce novel choices for several of the taxonomy dimensions, derive new histogram types by combining choices in effective ways and show how sampling techniques can be used to reduce the cost of histogram construction) presenting empirical results of its performance. This paper [22] extracts an accurate histogram from the dynamic data structure. Paper [23] discusses the efficiency of several estimation techniques including Equi-width, Equi-depth, the Rectangular Attribute Cardinality Map (R-ACM) and the Trapezoidal Attribute Cardinality Map (T-ACM), with empirical results. The reader is referred to [24], a paper that covers the rich and long history of using histograms for estimating selectivity factors.

Magic sets rewriting is a well-known heuristic optimization for complex decision support queries [25] [26]. There can be many variants of this rewriting even for a single query, which differ greatly in execution performance. [27] proposes cost-based techniques for selecting an efficient variant from the many choices.

A large body of works focus on reordering joins to find the lowest-cost join order, because it is well known that join order has a major effect on query performance. [28] analyses optimizers from three classes: heuristic, randomized and genetic algorithms. Along the years different techniques and combinations of them have been developed around the idea of join reordering. Some work [29] has focused on emphasizing optimization of a single select-project-join query in centralized relational databases. [30] also investigates the problem of optimizing Select—Project—Join queries with large numbers of joins, guided on heuristics. A different approach is proposed in [31], which implements a top-down join enumeration algorithm.

Klug et al. [32] work is centered on query equivalence for aggregate functions, formally defining aggregate functions and extendig relational calculus to include aggregate functions in a natural manner. There are also several papers that study equivalence and minimization of unaggregated SQL queries with equality comparisons and possibly with sub-queries, for example [33], that follows the approach of [34], in which the study concentrates on Datalog translations of such queries, that is on combined-semantics conjunctive queries.

The optimization techniques mentioned along this section are implemented in commercial and open source database systems. However, experience shows that even sophisticated database optimizers are often not capable of successfully optimizing provenance computations produced by rewrite-based provenance systems such as *Perm* [6], *DBNotes* [8] and *GProM* [14].

Grust et al. [35] also faces a similar challenge, even though this work is for a different application domain: Translating XQuery (an XML query language) int SQL. Similar to our work this paper proposes to use a heuristic optimizer which rewrites SQL code, in their case an implementation of an XQuery query, before sending it to the database. Some of the rewrite rules introduced in this work may also be applicable to our domain. However, given the different application (provenance computation vs. XQuery) the operator structure of the queries we produce differ significantly from theirs. Thus we had to investigate additional rewrite rules targeted for provenance computations, presented in this work. We also integrate these ideas with cost-based optimization.

CHAPTER 4
HEURISTIC OPTIMIZATION

As shown before, a query can be represented as a tree data structure. Operations are at the interior nodes and data items are at the leaves. The heuristic optimizations we use in GProM are applied to the relational algebra (AGM) representation of a query, that is, we apply heuristic rewrites to the AGM graph after it has been rewritten by the provenance rewrite module. In the case that the cost-based optimization is deactivated then the result of the heuristic optimizations is passed to the SQL code generators and executed on the back-end database. In case we activated the cost-based optimizer this module would take the algebra tree produced form the heuristic optimizer and produce a set of equivalent plans. It will estimate the cost for each plan and return the one with the lowest cost. It is of critical importance that the cost-based optimizer makes the ”right” choice on an execution.

We present each rule as $\frac{pre}{q \to q'}$ which is read as “If condition *pre* holds, then q can be rewritten as q'”. It is essential that all rules preserve equivalence (as detailed in Section 2.3) i.e., the input algebra expression is equivalent to the output algebra expression.

In the following sections we briefly describe the rules supported by our current implementation of the heuristic optimizer and motivate why they are useful in the context of our application. In addition to the rules we are also going to introduce Property Inference, which allows us to infer specific properties for the operators of an algebra expression. These properties can be used to simplify the heuristic rules.

4.1 Property Inference

Before applying any of the rewrite rules, we infer some properties that hold for operators of an algebra graph. These properties help us simplify the definition and application of the optimization rewrite rules. We use bottom-up and and top-down traversal of algebra trees to compute these properties for each operator in a query. Some of the properties can be computed in a single top-down or bottom-up traversal, whereas others require both.

The properties we consider in this work include:

- *Set*: set is a boolean property generated top-down. Its value indicates if the ancestors of an operator in the graph undergo duplicate elimination. Initially, set is true for all operators except for the root. We use this property to remove unnecessary duplicate removal operators.

- *Keys*: keys is the set of candidate keys of an operator generated bottom-up. It is used in the duplicate removal heuristic rule. For example, consider a relation R with schema $\{A, B, C, D\}$. If each tuple can be uniquely identified by attribute $\{A\}$ and by attributes $\{B, C\}$, then the keys property would be $\{\{A\}, \{B, C\}\}$.

- *EC*: EC is the short form of Equivalence Class and it is generated by a bottom-up followed by a top-down traversal. This property is a set of sets where each set contains attributes (and/or constants) that are guaranteed to have the same value in the output of an operator. For example, consider a relation R which has attributes $\{A, B, C, D\}$ and $EC = \{\{A\}, \{B, C\}, \{D\}\}$. We know that B and C are the same equivalence class which means B and C have the same value in each tuple of R.

- *Icols*: This property records which attributes are needed by the ancestors of an operator $\diamond$. For example, if relation R has attributes $\{A, B, C, D\}$, in the relational algebra expression $\Pi_{A,B}(R)$ we would set $icols(R) = \{A, B\}$, because the projection operator (the only parent operator of R) needs attributes $\{A, B\}$ to compute its result. We mainly use *icols* to determine redundant computations (e.g., in $\Pi_A(\Pi_{A,B+C\rightarrow D}(R))$ the computation $B + C$ is never used and, thus, can be omitted) and push projections (remove unneeded attributes early on to reduce the size of tuples).

Throughout the development of the Optimizer project, I have implemented the Key Property module. This is the reason why in the thesis I focus on this module, dedicating the following section; $key(op)$ denotes the candidate keys for an operator op. We describe how we implement this property for each of the operators, motivating its usefulness and proving some of the rules. We also include some explanatory examples on how this property is inferred over the algebra tree.

4.2 Keys Property

The Keys property of an operator stores which attributes in the result of the operator are guaranteed to uniquely identify each tuple of a relation. Each key k is represented as a set of columns $k = \{a_1, .., a_n\}$ and originates from a base-table constraint or can be caused by aggregation or duplicate removal operators.

This property is computed bottom-up, meaning we will always start computing the keys from the leaves of the tree and finish with the root. Rules work locally for single operators of a query, allowing us to compute this property for one operator at a time, without the need of a wider implementation.

Table 4.1 shows the inference rules for each operator that is supported in our optimizer. In the following we will discuss these rules thoroughly, but first we need to introduce an auxiliary function: Remove redundant keys $RRK(e)$ from an expression e.

After applying any of the following rules we will use the function $RRK(e)$ to make sure there do not exist any redundant keys. A particular key k_1 is redundant if it contains another key k_2. In this case k_1 could be removed. For a better understanding

Operator $\diamond$	Inferred *keys* property of $\diamond$
R	$key \leftarrow \{PK(R)\}$
$\{t\}$	$key \leftarrow \{\{a\} \mid a \in SCH(t)\}$
$\pi_{e_1 \leftarrow a_1,..,e_n \leftarrow a_n}(R)$	$key \leftarrow \{\{a_i \mid b \in k \wedge b = e_i\} \mid k \in R.key, k \subseteq \{a_1,..,a_n\}\}$
$\delta(R)$	$key \leftarrow R.key \cup \{cols(R)\}$
$R \bowtie_\theta S$	$key \leftarrow \{k_1 \cup k_2 \mid k_1 \in R.key, k_2 \in S.key\}$
$R \cup S$	$key \leftarrow \{\emptyset\}$
$R \cap S$	$key \leftarrow RRK(R.key \cup S.key[SCH(S) \leftarrow SCH(R)])$
$R - S$	$key \leftarrow R.key$
${}_G\gamma_A(R)$	$key \leftarrow \begin{cases} \{SCH(R)\} & \text{if } G = \emptyset \\ \{R.key \cap SCH(R)\} & \text{else} \end{cases}$
${}_G\omega_A(R)$	$key \leftarrow R.key$
$\sigma_{A=B}(R)$	$key \leftarrow RRK(R.key \cup \{k[B \leftarrow A] \mid k \in R.key\} \cup \{k[A \leftarrow B] \mid k \in R.key\})$

Table 4.1: Bottom-up inference of *keys* property for operator $\diamond$

of how this is used see the examples below, where we use this function.

4.2.1 Description for each operator

Table Access Operator

This operator returns relation R unmodified, for that reason the keys remain the same as in the input.

Example. *Consider a relation R with attributes (A, B, C) and the primary key $\{A\}$. After running the property inference, the Key property for the relation R will be set as $\{\{A\}\}$ ($R.key = \{PK(R)\} = \{\{A\}\}$).*

Constant Relation Operator

The constant relation operator results in a new relation with only one tuple, therefore every attribute will be a key (there is no possibility that there exist repeated tuples as we only have one).

Example. *Consider relation student from database 2.1. If we apply this operator to select tuple 2, the result of the query will only include one tuple containing (1222, Robert, Johnson, 3.8) and the keys will be all the attributes* $\{\{s\_id\}, \{s\_fname\}, \{s\_lname\}, \{s\_gpa\}\}$*, as there is only one tuple and there can not be any repeated values.*

Projection Operator

As a projection operator does not necessarily maintain all the attributes from the input relation (it usually only maintains some of them), before we can confirm a set of attributes as a key, we need to figure out whether each attribute contained in the key is part of the new relation.

Example. *Lets make it more clear with an example. Consider the relation S with attributes* $\{D, E, F\}$ *and candidate keys* $\{\{D\}, \{E, F\}\}$*. If we apply the query* $\pi_{E,F}(S)$ *our output table will only have one key* $\{\{E, F\}\}$*, as the other key* $\{\{D\}\}$ *is no longer part of the relation.*

Duplicate Removal Operator

The duplicate removal operator removes any duplicate tuple a relation may have. This way, when we apply this operator, besides preserving the keys from the input relation, we will also have a key which is all the attributes in the output, since all the tuples in the new relation will be unique (as seen in the operator definition). After inferring the key property, we will need to apply the auxiliary function to remove redundant keys $RRK(key(\delta(R)))$. In case we had keys in the input relation we can confirm that the output will have redundant keys (the keys from the input will be contained in the new key with all attributes).

Example. *See the following example for a better understanding of the previous explanation. Consider a new relation T with attributes $\{H, I\}$ which has some duplicate tuples. Since there are repeated tuples there is no primary key in this relation. If we apply the duplicate removal operator $\delta(T)$ we will obtain a relation without duplicates, and the schema of this relation will be the new key ($T.key = \{\{H, I\}\}$).*

In case we were applying the duplicate removal operator to the relation R (which has keys $\{\{A\}\}$), the keys we would obtain at first would be: $\{\{A\}, \{A, B, C\}\}$. Since $\{A\}$ is contained in the new key $\{A, B, C\}$ we will use the $RRK(R.key)$ function to make sure we do not have any redundant keys, removing the ones that contain other keys. The final keys we would obtain would be the same ones we had in the input ($\{\{A\}\}$).

Join Operator

The join operator returns all combinations of tuples from R and S that match the condition θ, so the resulting attributes will be the same attributes we had separately in each of the input relations. If we union the sets of keys, we will be combining the keys from each side of the input, assuring the new sets of attributes correspond to keys of the output relation.

Example. *Let's give an example joining the relations R and S introduced in the previous examples. We can use this query: $R \bowtie_\theta S$. Relation R has keys: $\{\{A\}\}$; and relation S: $\{\{D\}, \{E, F\}\}$. The result of this query will consist of the combination of the tuples from both relations that satisfy the condition θ. As we can see, the output relation will maintain all the attributes, and the keys will be a combination of the previous ones: $\{\{A, D\}, \{A, E, F\}\}$.*

Union Operator

$R \cup S$ will return all sets of tuples contained in R, S or both relations. We can not guarantee whether there will or will not be duplicates in the output relation. That is why we set the output relation keys to empty.

Example. *Consider a simple example that demonstrates how this works. We union a relation R with another relation S. The output relation will contain all tuples in R and S. Even when both input relations have keys, there may still be duplicates in the new relation (or at least we can not guarantee that there are not). Then there will not exist any unique attributes, which forces us to set the keys to the empty set.*

Intersection Operator

The intersection operator returns tuples which are present in both input relations. Considering the intersection operator definition we can state that if a specific set of keys was valid in the input relation it will also be in the output relation (as the output relation will only consist of tuples which were present in both input relations).

Example. *For example, we intersect the previously introduced relations R (with key $\{\{A\}\}$) and S (with keys $\{\{D\},\{E,F\}\}$): $R \cap S$. The output relation has the attributes $\{A,B,C\}$ (with tuples present in S renamed). Then the keys will be preserved from both sides of the input: $\{\{A\},\{B,C\}\}$. Observe that the first key corresponds to the key from relation R, the second and third to the keys from relation S renamed. We will have to apply the function to remove redundant keys to make sure we do not have keys containing other keys. After cleaning the list we will have our final keys: $\{\{A\},\{B,C\}\}$.*

Difference Operator

As the difference definition implies, it only returns tuples present in the left input (R). That makes it obvious that the output keys will only be those present in the left input.

Example. *Following with the same example, let us consider the query: $R - S$. The resulting relation will only have the tuples from R which are not present in S. Therefore the keys will be the same they were in the input relation R: $\{\{A\}\}$.*

Aggregation Operator

For the aggregation operator we need to consider two different cases: (1) When there is a *GroupBy* clause, the returned tuples will be grouped by the attributes, assuring that these will be unique and candidate keys. (2) In case there is no *GroupBy* the query will only return one tuple, then it would be the same case as for the constant relation operator.

Example. *Consider the relations student and enrrollment from the database 2.1. We can use a query to count all the students with an `id_gpa` equal or greater than* 3.5*. This will return one tuple with the number of students that have this gpa (`gpa=>3.5`), therefore the key will be the attributes for this tuple: $\{\{gpa => 3.5\}\}$. In case we have the GroupBy clause, count the number of students in each course, grouping them by the `course_id`. We might have the same number of students in each course, so this attribute can not be a key. But as we are grouping them by the course id, the values for this attribute will be unique, resulting as an output key for the query: $\{\{course\_id\}\}$.*

Window Operator

The window operator performs a calculation across a set of tuples that are somehow related to the current tuple. As we are not making any modification to the input, we simply return the keys from the input.

Selection Operator

Selection $\sigma_{A=B}(R)$ returns a relation containing all and only the tuples of R that fulfill the condition, so only tuples present in R will later be present in the output relation. Therefore the keys will remain the same ones we had in the input, adding some in case one side of the condition is a key. To simplify the rule we are assuming the condition is $A = B$, which would also cover a conjunction of comparisons ($A = B \land B = C$).

Example. *For example, we make a selection in the previous relation R where $A = B$. The output relation will include all those tuples from R that fulfill the condition. Therefore the keys will remain the same and add* `B` *as a new key: $\{\{A\},\{B\}\}$. IN this case* `A` *is a key and $A = B$, then we can assure that* `B` *will also be key.*

Other Operators

We have also implemented the key property inference for some operators, which are not explained in detail as they are not as prevalent as the previous ones. For the order operator we will keep the same keys from the input. For the json table operator we set the keys to an empty set and for the nesting operator we will return the keys from the left input.

4.2.2 Keys Property Usage

Keys are useful because they allow us to easily simplify a query, reducing its computation cost considerably. There exist a variety of use cases which would benefit from the key property. However, we have focused our work on removing Duplicate Removal operators. Additional rules are left for future work.

$$\frac{keys(R) \neq \emptyset}{\delta(R) \rightarrow R} \tag{4.1}$$

The rationale behind Rule (4.1) is that if relation R has at least one candidate key, then it cannot contain duplicates because the values of the key columns are unique in R (definition of a super-key). Thus, the Duplicate Removal operator has no effect and we can safely remove it.

Formal Analysis

Assuming K is a key from R, we can state from the Key definition that: $\forall t^n \in \pi_K(R) \rightarrow n = 1$. Knowing that the duplicate removal operator is defined by $\delta(R) = \{t^1 \mid t^n \in R \ \wedge n \neq 0\}$. Then if K is a key, $n = 1$. This leads us to affirm that in the definition of the duplicate removal operator $t^1 = t^n$ because $n = 1$. We can confirm that the duplicate removal will not have any affect as it is the same query before and after the operator.

Example. *For example, consider the relational algebra expression $\delta(R)$ and the keys of R are $\{\{A\}\}$. Here each tuple in relation R is unique. Thus the duplicate removal operator has no effect, we can remove it. Then $\delta(R) \rightarrow R$.*

Some other rules that we have considered and could be implemented in future works include Join removal and Duplicate Removal move around. Both rules have been introduced by Grust et al. in [35]. These rules are based on the idea of the *equi-joins* introduced by the compilation rules. The Duplicate Removal move around is based on the fact that introducing such operator at the top of the tree graph does not alter the result of a query. This together with the rule that removes Duplicate Removal operators is what is called Duplicate Removal move around. The Join removal rule is more complex to describe, but the reader is referred to paper [35] for a detailed description.

Example 6. *Consider the following algebra expression over the database from Table 2.1: We will use the same example from Section 2.2, inferring the Key property throughout the sequence of trees.*

$$\begin{aligned} j &= student \bowtie_{s\_id=student\_id} enrollment \\ q_{ex} &= \pi_{s\_fname,s\_lname}(\sigma_{course\_id='123'}(j)) \end{aligned} \tag{4.2}$$

As explained in the previous sections, the Key property is inferred bottom-up, starting at the table access operator. The student relation key is $\{\{s\_id\}\}$, *and for the enrollment relation is* $\{\{student\_id, course\_id\}\}$. *Observe that we represent the keys (together with the relation or operator) in the tree graph for a relation or an operator like:* $student\{\{s\_id\}\}$.

In Figure 4.1, graph (a), we can see that both relations have a primary key. We will compute the join operator as a cross product followed by a selection, as it is described in its definition ($student \bowtie_{s\_id=student\_id} enrrollment \equiv \sigma_{s\_id=student\_id}(student \times enrrollment)$*). In the following graph (b) we can see how we have computed the key property for the join (cross product) operator. As described before, the join operator has as key the combination of keys from relations* $students$ *and* $enrrollment$. *The keys* *for this operator will be the combination of* $\{\{s\_id\}\}$ *with* $\{\{student\_id, course\_id\}\}$*, resulting in* $\{\{s\_id, student\_id, course\_id\}\}$*. Then, in figure (c) we compute the selection* $\sigma_{s\_id=student\_id}$ *from the join together with the* $course\_id =' 123'$ *selection and the keys are* $\{\{s\_id, course\_id\}, \{student\_id, course\_id\}\}$*.*

When we compute the projection, we are selecting only some specific attributes from the input relation, which brings us the need to redefine the previous keys, making sure they still exist. In this case, we are only projecting the name and last name attributes, which are not a key. Therefore the projection operator will have the key set empty.

4.2.3 Implementation

Table Access Operator

The key inference for a table access operator simply takes the primary keys from the input relation, as there is no modification.

Constant Relation Operator

For the $\{t\}$ operator we simply take all the attributes from the input (using the auxiliary function $SCH(R)$) and set them individually (because we want the minimal keys) as the key.

Projection Operator

To infer the Key property for the projection operator we will first map the output relation attributes enumerating their position. This will be useful to check in each set of keys k, if the attributes of k are projected in the output relation (attributes exist in the map). We will preserve all those keys which the attributes are contained in the map previously created.

Duplicate Removal Operator

To explain the key property inference for the duplicate removal operator $\delta(R)$ we rely on the auxiliary function $SCH(R)$ introduced in Section 2.1 that determines the attributes of a relation. For this operator the key inferred will be a union between the input keys and all the attributes ($SCH(R)$) of the output. That is, in case there are no keys in the input, the output will have all the attributes as key. After inferring the key property, we will need to apply the auxiliary function to remove the possible redundant keys $RRK(\delta(R))$.

Join Operator

For the join operator $R \bowtie_{\theta} S$ the inferred keys will be obtained from combining all the sets of keys from relation R with all the sets of keys from relation S.

Union Operator

The union operator $R \cup S$ has the simplest implementation as the keys inferred is an empty set, as we can not guarantee whether the output relation will not contain duplicates.

Intersection Operator

For the intersection operator $R \cap S$ we will rename the attributes of the right input (S) and then infer the keys from both inputs (R and renamed S).

Difference Operator

The difference operator $R - S$ returns the keys from the left input, as only tuples present in the left input will be present in the output relation.

Aggregation Operator

For the aggregation operator ${}_G\gamma_A(R)$ we have to consider two different cases: (1) When there is a *Group By* clause then we will intersect the keys from the input with the group by G attributes, (2) In case there is no G clause, we will simply return all the attributes from the output.

Window Operator

The window operator is easily implemented, as we simply take the keys from the input.

Selection Operator

The selection operator has a simplified implementation, which is taking the same keys as in the input and then using the EC property to remove any redundant keys.

4.2.4 Formal Analysis

In this section we prove the mathematical key property inference for two different operators: Constant relation and duplicate removal.

Constant Relation Operator

The constant relation operator has as key every attribute in the relation independently ($\{\{a\} \mid a \in SCH(t)\}$ is key K). Then using the key definitions K is key if $\forall t^n \in \pi_K\{t\} \rightarrow n = 1$.
Assuming $n > 1$, with the Constant Relation Operator definition $\{t\} = \{t'\}$ The constant relation operator definition implies that n can not be greater than 1. This contradiction leads us to confirm that every attribute in the relation is key.

Duplicate Removal Operator

As stated before, for the duplicate removal operator the keys come from a union of
(1) All the attributes in the output relation, that is $SCH(\delta(R))$ is a key K and
(2) The keys from the input, this is $K \in key(R)$ is a key.
Afterwards we apply the $RRK()$ function to make sure we do not have any redundant keys. To prove the computations of the key property for this operator, we will prove each of the previous statements separately, using the key definition in section 2.1:

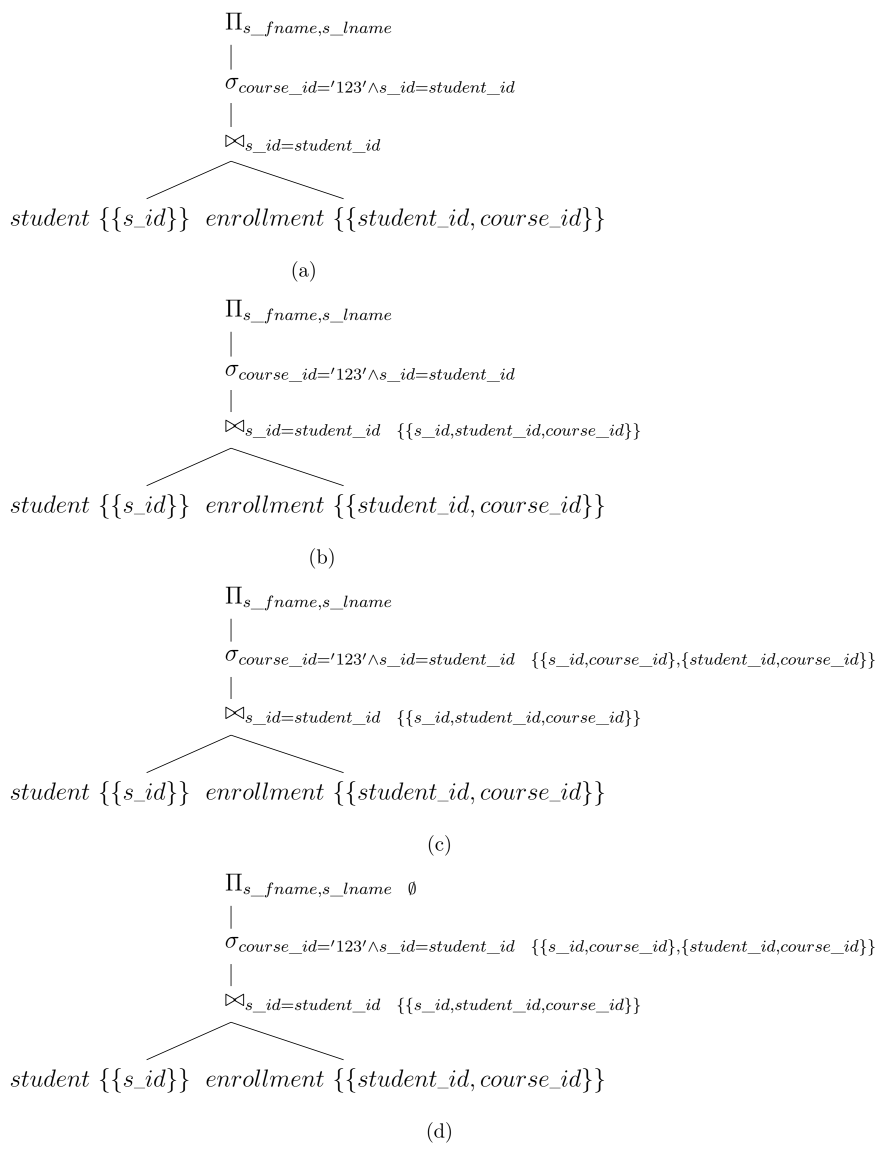
(1) All attributes are a key: $SCH(\delta(R))$, K is a key $\forall t^n \in \pi_{SCH(\delta(R))}(\delta(R)) \rightarrow n = 1$. This can be rewritten as $\forall t^n \in \delta(R) \rightarrow n = 1$
Assuming that exists t with $n > 1$, $t^n \in \delta(R) \rightarrow n = 1$. As shown in the background section, the duplicate removal operator definition is $\delta(R) = \{t^1 \mid t^n \in R \ \wedge n \neq 0\}$.

That means that the tuples for this operator have multiplicity 1 (t^1). This leads to a contradiction with the definition of δ operator, which proves that the schema from R is a key.

(2) We can affirm that: $\forall t^n \in q \Rightarrow t^m \in \delta(q) \rightarrow m < n$. This means $\pi_K(\delta(q)) \subseteq \pi_K(q)$. Then $t^n \in \pi_K(\delta(q)) \Rightarrow t^m \in \pi_K(q) \rightarrow m > n$. From the key property we know n has to be 1, then m would need to be greater than 1. We conclude that K can not be a key in this case, which is a contradiction because K is a key from the input.

4.3 Application of Heuristic Rewrites

Right now we use a manually determined fixed order for applying these rules to an input query which is already quite effective. In the future we would like to replace it by a fix-point computation. However, this requires us to prove that the fix-point computation is correct first. Also we would like to explore additional rules such as moving window operators.



Π_{s_fname,s_lname}
$\sigma_{course\_id='123' \wedge s\_id=student\_id}$
$\bowtie_{s\_id=student\_id}$
$student\ \{\{s\_id\}\}$
$enrollment\ \{\{student\_id, course\_id\}\}$
(a)
Π_{s_fname,s_lname}
$\sigma_{course\_id='123' \wedge s\_id=student\_id}$
$\bowtie_{s\_id=student\_id}\ \{\{s\_id,student\_id,course\_id\}\}$
$student\ \{\{s\_id\}\}$
$enrollment\ \{\{student\_id, course\_id\}\}$
(b)
Π_{s_fname,s_lname}
$\sigma_{course\_id='123' \wedge s\_id=student\_id}\ \{\{s\_id,course\_id\},\{student\_id,course\_id\}\}$
$\bowtie_{s\_id=student\_id}\ \{\{s\_id,student\_id,course\_id\}\}$
$student\ \{\{s\_id\}\}$
$enrollment\ \{\{student\_id, course\_id\}\}$
(c)
$\Pi_{s\_fname,s\_lname}\ \emptyset$
$\sigma_{course\_id='123' \wedge s\_id=student\_id}\ \{\{s\_id,course\_id\},\{student\_id,course\_id\}\}$
$\bowtie_{s\_id=student\_id}\ \{\{s\_id,student\_id,course\_id\}\}$
$student\ \{\{s\_id\}\}$
$enrollment\ \{\{student\_id, course\_id\}\}$
(d)

Figure 4.1: Example Key Property Computation

CHAPTER 5
EXPERIMENTS

My experimental evaluation is focused on studying the performance improvements gained by using heuristic optimization, specifically the usage of the Key Property and the rules that take advantage of it. All expermients were executed on a machine with the configurations shown in Table 5.1.

To evaluate the efficiency of our optimizer, specially the use of the key property, we will perform our tests with game provenance queries. This kind of provenance will highlight the contribution of inferring the key property to the operators to facilitate removing duplicate removal operators.

Game Provenance is defined by viewing the evaluation of a first-order query as a game between two players who argue weather a tuple is in the query answer. The game provenance query is much more complicated with many more rules and levels, meaning that there exist many levels of rules which use the head of other rules in their body. It also reuses a lot of expressions. Duplicate removal operators are introduced to achieve the set semantics required by the Datalog, the query language in which the game provenance computations is originally expressed in before the translation into relational algebra. This results in many more levels of duplicate removal operators than in the input query.

That is why it becomes more important to remove these duplicate removal

Name	Type
CPUs	2 x AMD Opteron 4238 (12cores)
RAM	128 GB
Hardware	4 x 1TB 7.2K HDs

Table 5.1: Experiment Configuration

operators and give the database the option to reorder the operators and select what to materialize. The Key Property is key in removing duplicate removals, for this reason we have chosen game provenance to demonstrate the efficiency of the optimizer and the key property.

5.1 Dataset

For these experiments we use TPC-H datasets. The TCP-H benchmark [36] is a standard decision support benchmark. It consists of a suite of business oriented ad-hoc queries and concurrent data modifications. The queries and the data populating the database have been chosen to have broad industry-wide relevance while maintaining a sufficient degree of ease of implement. This benchmark provides large volumes of data and queries with a high degree of complexity. We generated TPC-H benchmark datasets of 100MB, 1GB and 10GB.

5.2 Workloads

To test the effectiveness of this property we run the same query with and without the optimizer. We use several database sizes (100MB, 1GB and 10GB) and repeat each experiment a number of times depending on the database size. For the 100MB we do 1000 iterations as the results are the most instable because of its small size. We do 100 iterations for the database of 1GB and 10 iterations for the 10GB sized database (the most stable results). This way we can see how the effectiveness changes according to the size of the database.

The datalog query for which we compute game provenance is the following:

$$"Q(C\_CUSTKEY, L\_LINENUMBER) :-$$
$$ORDERS(L\_ORDERKEY, C\_CUSTKEY, a, b, ...),$$
$$LINEITEM(L\_ORDERKEY, c, L\_LINENUMBER, d, ...).$$
$$QT(C\_NAME, L\_LINENUMBER) :- Q(C\_CUSTKEY, L\_LINENUMBER),$$

$$CUSTOMER(C\_CUSTKEY, C\_NAME, e, f, ...).$$
$$WHY(QT('Customer\#000000001', 1)).”$$

First we join relations *Order* and *Lineitem* and the resulting relation is joined with *Customers*. The result of this query is a relation that relates the Customer Name with the Linenumber from the relation Lineitem.

The parameters we use to measure the effectiveness of the optimizer are: *Minimum value*, *Percentil* 0.05, *Percentil* 0.25, *Median*, *Percentil* 0.75, *Percentil* 0.95, *Maximum value* and *Standard deviation*.

5.3 Results

We provide three graphs (on for each database size) comparing the performance using and not using the optimizer. The graphs clearly represent the difference in performance when not using an optimizer (red histogram) and using one (blue histogram).

In Figure 5.1 we can see that the performance is improved, but in general terms it does not reach the 50%. This is the most unstable test because of the small size of the database and that is why we do 1000 iterations This way we can figure out the tendency of the run time with and without the optimizer more accurately. We also notice that the maximum value can rise uncontrollably when not using an optimizer, reaching a time performance of 12s. When using an optimizer on a bigger database size we have a more controlled environment for performing the query resulting in a lower peak time and more consistent results.

As mentioned before, the bigger the database size is the more stable results will be, and we will need less iterations to have accurate and reliable results. With the 1GB and 10GB databases we will be able to extrapolate the results with less iterations. The graphs clearly represent how the stability of the tests increases with the size of the database, as the peaks are not as noticeable. We also observe that

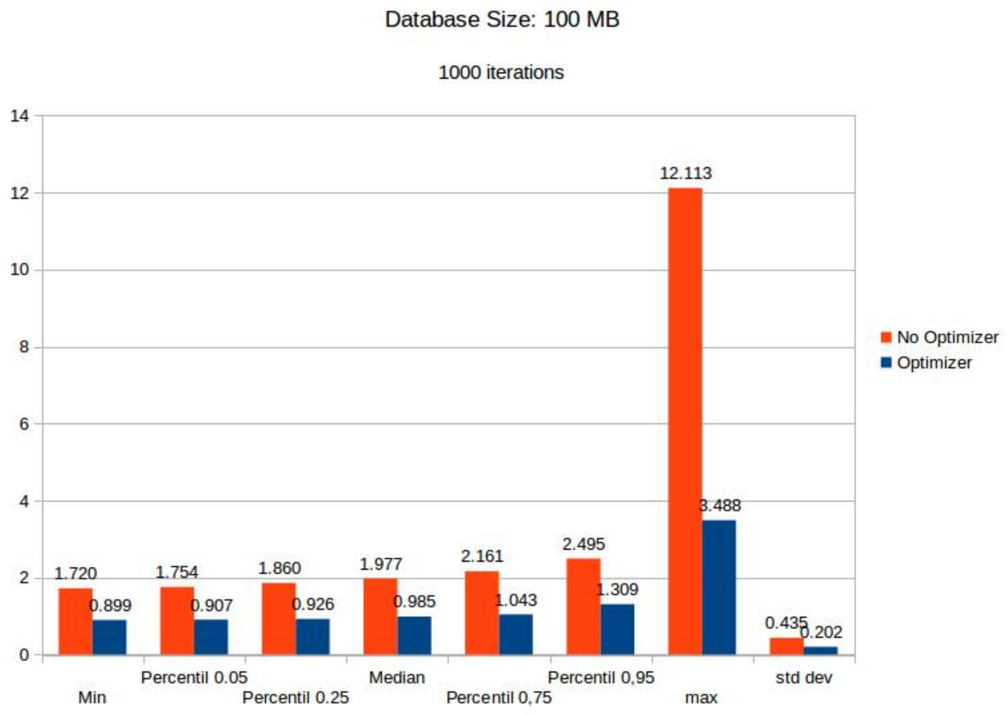


Figure 5.1: Game Provenance Query - 100MB

both of this tests result in a more significant performance improvement reaching up to 60%. This leads us to state that the optimizer is working and that it makes a greater difference when the database size increases.

We conclude that our experiments confirm the effectiveness of the Key property in the heuristic optimizer, improving the performance up to 60%.

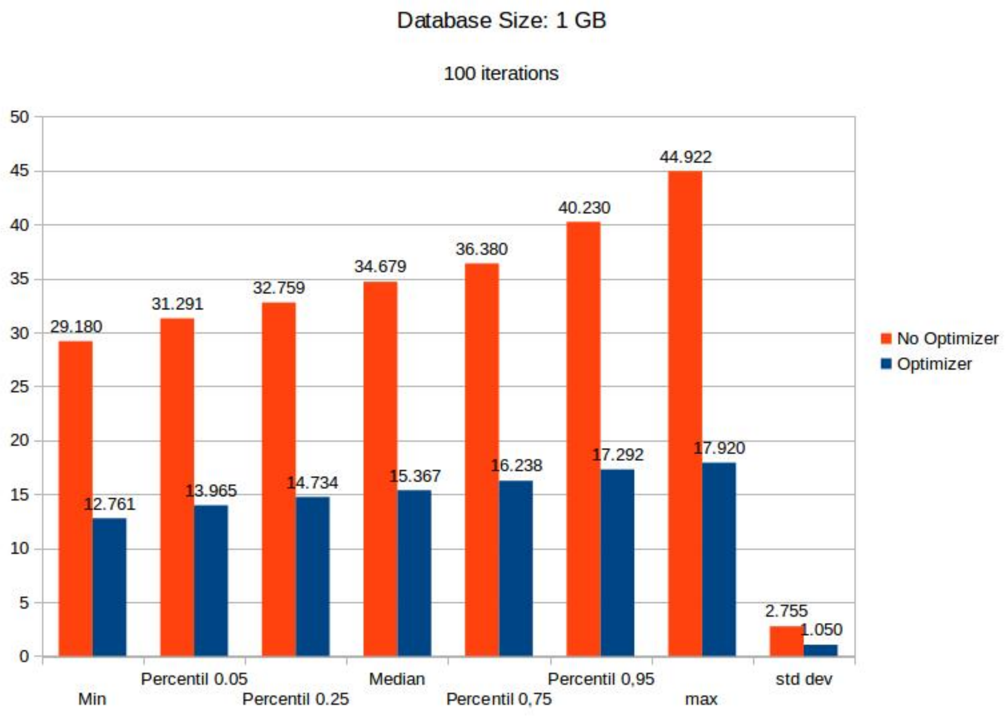


Figure 5.2: Game Provenance Query - 1GB

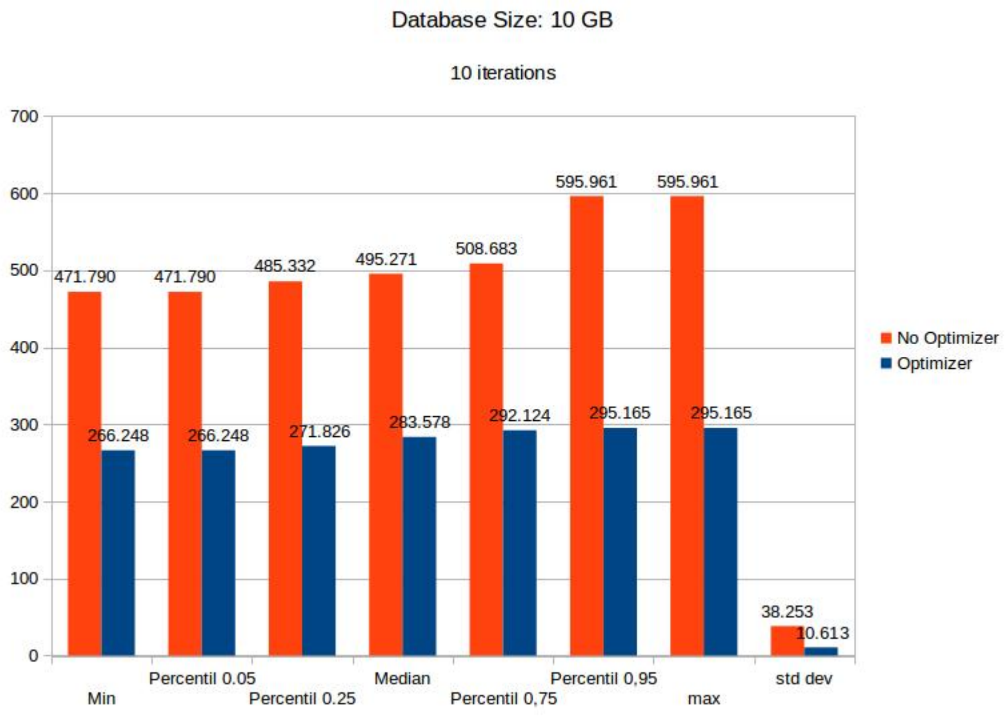


Figure 5.3: Game Provenance Query - 10GB

CHAPTER 6
CONCLUSIONS AND FUTURE WORK

Throughout this thesis I have presented heuristic optimization techniques for queries produced by rewrite-based provenance approaches and discussed the implementation of these techniques in the GProM system. The motivation of this work is the realization that queries that compute provenance are often not successfully optimized, not even by sophisticated database optimizers. This is because of their complexity and unusual structure.

For heuristic optimization, we have implemented some rules which significantly improve performance, in some cases turning queries with billions of years of runtime into queries that run in milliseconds. Specifically I have implemented the Key Property, which allows rules such us removing duplicate removal operators to be more effective. The Keys property of an operator stores which attributes in the result of the operator are guaranteed to uniquely identify each tuple of a relation. We infer this property to each one of the operators in a query, bottom-up.

My experimental results confirm that, the rule that removes duplicate removal operator using the key property, significantly improves performance. We have proved this by using a game provenance query. These queries introduce many levels of duplicate removal operators to achieve the set semantics of the Datalog, which is later translated into relational algebra. We have seen that in these cases the removal of the duplicate removal operators is key in improving performance, as it allows the database to reorder the operators increasing efficiency.

There are several interesting themes of future work. The obvious next steps include more extensive experiments, implementations of additional heuristic rules, and formally proving competitiveness and correctness of the algorithms used. Additional heuristic rules that would follow the work I have done, would be implementing the rules introduced in Section 4. This two rules, Join removal and Duplicate Removal move around, would be implemented using the Key Property module.

Provenance can be quite large, even for queries with a small result size. Related work has explored compressed representations of provenance. It would be interesting to see how such techniques can be integrated with provenance computation and see how they interact with our optimizer. Furthermore, it would be engaging to explore more sophisticated methods for applying heuristic rules and study their applicability.

BIBLIOGRAPHY

[1] Grigoris Karvounarakis and Todd J Green. Semiring-annotated data: queries and provenance? *ACM SIGMOD Record*, 41(3):5–14, 2012.

[2] Sven Köhler, Bertram Ludäscher, and Daniel Zinn. First-order provenance games. In *In Search of Elegance in the Theory and Practice of Computation*, pages 382–399. Springer, 2013.

[3] Edgar F Codd. *The relational model for database management: version 2*. Addison-Wesley Longman Publishing Co., Inc., 1990.

[4] Peter Buneman, Sanjeev Khanna, and Tan Wang-Chiew. Why and where: A characterization of data provenance. In *Database Theory—ICDT 2001*, pages 316–330. Springer, 2001.

[5] Yingwei Cui and Jennifer Widom. Lineage tracing for general data warehouse transformations. *The VLDB Journal—The International Journal on Very Large Data Bases*, 12(1):41–58, 2003.

[6] Boris Glavic. *Perm: Efficient Provenance Support for Relational Databases*. PhD thesis, University of Zurich, 2010.

[7] Charu C Aggarwal. Trio a system for data uncertainty and lineage. In *Managing and Mining Uncertain Data*, pages 1–35. Springer, 2009.

[8] Laura Chiticariu, Wang-Chiew Tan, and Gaurav Vijayvargiya. Dbnotes: a post-it system for relational databases based on provenance. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 942–944. ACM, 2005.

[9] Todd J Green, Grigoris Karvounarakis, Zachary G Ives, and Val Tannen. Provenance in orchestra. 2010.

[10] James Cheney, Laura Chiticariu, and Wang-Chiew Tan. *Provenance in databases: Why, how, and where*, volume 4. Now Publishers Inc, 2009.

[11] Dan Olteanu and Jakub Závodnỳ. On factorisation of provenance polynomials. In *USENIX TaPP Workshop*, 2011.

[12] Yael Amsterdamer, Daniel Deutch, Tova Milo, and Val Tannen. On provenance minimization. *ACM Transactions on Database Systems (TODS)*, 37(4):30, 2012.

[13] Adriane P Chapman, Hosagrahar V Jagadish, and Prakash Ramanan. Efficient provenance storage. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 993–1006. ACM, 2008.

[14] Bahareh Arab, Dieter Gawlick, Venkatesh Radhakrishnan, Hao Guo, and Boris Glavic. A generic provenance middleware for database queries, updates, and transactions. In *Proceedings of the 6th USENIX Workshop on the Theory and Practice of Provenance (TaPP)*, 2014.

[15] M. Jarke and J. Koch. Query Optimization in Database Systems. *ACM Computing Surveys (CSUR)*, 16(2):111–152, 1984.

[16] Morton M. Astrahan, Mike W. Blasgen, Donald D. Chamberlin, Kapali P. Eswaran, JN Gray, Patricia P. Griffiths, W Frank King, Raymond A. Lorie, Paul R. McJones, James W. Mehl, et al. System r: relational approach to database management. *ACM Transactions on Database Systems (TODS)*, 1(2):97–137, 1976.

[17] P.G. Selinger, MM Astrahan, DD Chamberlin, RA Lorie, and TG Price. Access path selection in a relational database management system. *Proceedings of the 1979 ACM SIGMOD international conference on Management of data*, pages 23–34, 1979.

[18] Robert Philip Kooi. The optimization of queries in relational databases. 1980.

[19] Gregory Piatetsky-Shapiro and Charles Connell. Accurate estimation of the number of tuples satisfying a condition. In *ACM SIGMOD Record*, volume 14, pages 256–276. ACM, 1984.

[20] M Muralikrishna and David J DeWitt. Equi-depth multidimensional histograms. In *ACM SIGMOD Record*, volume 17, pages 28–36. ACM, 1988.

[21] Viswanath Poosala, Peter J Haas, Yannis E Ioannidis, and Eugene J Shekita. Improved histograms for selectivity estimation of range predicates. *ACM SIGMOD Record*, 25(2):294–305, 1996.

[22] Nitin Thaper, Sudipto Guha, Piotr Indyk, and Nick Koudas. Dynamic multidimensional histograms. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 428–439. ACM, 2002.

[23] B John Oommen and Luis G Rueda. The efficiency of histogram-like techniques for database query optimization. *The Computer Journal*, 45(5):494–510, 2002.

[24] Yannis Ioannidis. The history of histograms (abridged). In *Proceedings of the 29th international conference on Very large data bases-Volume 29*, pages 19–30. VLDB Endowment, 2003.

[25] Francois Bancilhon, David Maier, Yehoshua Sagiv, and Jeffrey D Ullman. Magic sets and other strange ways to implement logic programs. In *Proceedings of the fifth ACM SIGACT-SIGMOD symposium on Principles of database systems*, pages 1–15. ACM, 1985.

[26] Inderpal Singh Mumick and Hamid Pirahesh. Implementation of magic-sets in a relational database system. In *ACM SIGMOD Record*, volume 23, pages 103–114. ACM, 1994.

[27] Praveen Seshadri, Joseph M Hellerstein, Hamid Pirahesh, TY Leung, Raghu Ramakrishnan, Divesh Srivastava, Peter J Stuckey, and S Sudarshan. Cost-based optimization for magic: Algebra and implementation. In *ACM SIGMOD Record*, volume 25, pages 435–446. ACM, 1996.

[28] Michael Steinbrunn, Guido Moerkotte, and Alfons Kemper. Heuristic and randomized optimization for the join ordering problem. *The VLDB Journal—The International Journal on Very Large Data Bases*, 6(3):191–208, 1997.

[29] Yannis E Ioannidis. Query optimization. *ACM Computing Surveys (CSUR)*, 28(1):121–123, 1996.

[30] Arun Swami and Anoop Gupta. *Optimization of large join queries*, volume 17. ACM, 1988.

[31] Pit Fender and Guido Moerkotte. A new, highly efficient, and easy to implement top-down join enumeration algorithm. In *Data Engineering (ICDE), 2011 IEEE 27th International Conference on*, pages 864–875. IEEE, 2011.

[32] Anthony Klug. Equivalence of relational algebra and relational calculus query languages having aggregate functions. *Journal of the ACM (JACM)*, 29(3):699–717, 1982.

[33] Rada Chirkova. Equivalence and minimization of conjunctive queries under combined semantics. In *Proceedings of the 15th International Conference on Database Theory*, pages 262–273. ACM, 2012.

[34] Sara Cohen. Equivalence of queries that are sensitive to multiplicities. *The VLDB Journal—The International Journal on Very Large Data Bases*, 18(3):765–785, 2009.

[35] Torsten Grust, Manuel Mayr, and Jan Rittinger. Let sql drive the xquery workhorse (xquery join graph isolation). In *Proceedings of the 13th International Conference on Extending Database Technology*, pages 147–158. ACM, 2010.

[36] "http://www.tpc.org/tpch".