CASSANDRA+: A SYSTEM FOR DOING QUERY-DRIVEN SIMULATIONS

P. Balasubramanian

Alexander Tuzhilin

Information Systems Department
Stern School of Business
New York University

Working Paper Series
STERN IS-93-42

Cassandra$^+$: A System for Doing Query-Driven Simulations

P. Balasubramanian Alexander Tuzhilin *

Information Systems Department
Leonard N. Stern School of Business

Abstract

A Query-Driven Simulation system *Cassandra*$^+$ is described that allows end-users to ask queries about outcomes of running simulation models written in various simulation languages. The architecture of Cassandra$^+$, its query language *SimQL*, and the implementation of the system are described. It is argued that Query-Driven Simulations provide a more declarative, flexible, and interactive approach to asking questions about simulation outcomes than the traditional approaches of letting the end-users run simulations and gather statistics on the trace files being produced.

KEY WORDS: Decision Support, Query-Driven Simulation, Discrete-Event Simulations, Temporal databases, Temporal Query Languages.

1 Introduction

Throughout their daily activities, decision makers in various organizations have to anticipate future events in order to make better decisions and adopt effective strategies for reaching desired goals. For example in a manufacturing organization, a foreman on the shop floor may want to know which machines will have a utilization ratio of more than 90% during the next two days, or a finance manager may want to know what will the average cost of producing jobs of type-A be, or a salesman needs to know if it is possible to complete job-5 in four days if it is assigned the highest priority. In the banking industry, a manager may want to know how much cash reserves will the bank have within a month's time. In the sea freight industry, a dockyard manager or a customs official may want to know what cargo will be delivered to his/her dockyard within the next week and in what quantities.

To deal with these types of questions in complex industrial and organizational settings, simulations are often used [LK91, BC84]. To answer questions such as the ones presented above, summary

*Address: 44 West 4th Street, Room 9-78, New York, NY 10012, e-mail: pbalasub@rnd.stern.nyu.edu, atuzhili@rnd.stern.nyu.edu

statistics are usually collected in one of the following two ways. In the first approach, summary statistics are computed inside the simulation program, and the program prints these statistics. In the second approach, various simulated events are recorded in the *trace files*, and then statistics are collected from these trace files by either writing programs in one of the programming languages, such as Fortran or C, or by using one of the statistical packages, such as SAS [SAS89]. We will call this traditional approach *simulate-and-gather-statistics (SAGS)* approach.

Since most of the people who ask questions about future outcomes of business processes in their organizations, such as a foreman, a salesman, or a bank manager, do not know much about simulations, programming languages, or statistical packages, they cannot ask ad-hoc questions about future outcomes of their business processes as the questions arise "on-the-fly." Instead, they have to rely on the systems developed by information systems departments that support a fixed set of "canned" questions. Clearly, this situation is unsatisfactory in many organizations, such as manufacturing, transportation, or in the military, where various users want to ask many different questions about simulation outcomes of various models [BT93].

In this paper, we describe the *Query Driven Simulations (QDS)* approach, that addresses this problem. QDS is an approach to simulations in which the user first asks queries about outcomes of simulations expressed in a *declarative* query language and then, depending on the query being asked, appropriate simulations are launched and events necessary to answer the query are recorded in the trace files. After the simulation runs are completed, the query is evaluated on the trace files(s) of events recorded by the simulation program. For example, assume an operations manager at a car manufacturing plant wants to know how many cars will be produced at the plant within the next week, and assume that he/she asks this query on the simulation model *Manufact-Model-3*. According to the QDS approach, it should be determined first what events in the *Manufact-Model-3* model should be traced and for how long in order to answer the query (i.e. the event *Finished(Car, Time)* should be traced for a week of simulated time). Then simulations are launched for that amount of time, the trace file for the event *Finished* is generated, and the query is answered based on the information contained in this trace file.

In the paper, we also present a specific QDS system, called *Cassandra*$^+$ and describe its architecture. As part of Cassandra$^+$, we describe the language *SimQL* for asking queries about simulation outcomes. We designed SimQL to make it simple enough for a non-technical user to understand and use it, and powerful enough to be used in complex simulations.

The QDS approach has the following advantages over the SAGS approach:

- *Declarativeness.* The user can ask a question in a declarative query language and does not

have to know any simulation and statistical packages, or write programs in conventional programming languages[1].

- *Flexibility.* QDS user can ask *any* query expressible in the query language of that system. This is in contrast to the fixed set of "canned" queries supported by the simulation specialist in the SAGS approach.

- *Interactiveness.* QDS user can ask queries "on-the-fly" as they arise without any help from a simulation specialist.

Query languages in the context of simulations were studied before. In [Len93], a database of simulation models, called a modelbase, was constructed based on the structured modeling approach [Geo87]. As part of the structured modeling approach, [Len93] uses the query language defined for this approach by Geoffrion[Geo87]. Although the system supports queries, these queries are used in a totally different context: they are used for asking questions about the models themselves (e.g. which models stored in the modelbase are manufacturing models), not about simulation traces produced by running the models.

In [MW89], Miller and Weyrich developed the SIMODULA system that has its own SQL-like query language (with object-oriented features added to it) for asking questions about simulations. Each model has a relation of input parameters and outcomes of previously executed simulations associated with that model. For example, a banking model may have a BankScenario relation associated with it that has input parameters, such as number of tellers, mean interarrival rate, mean service time, and the output parameters, such as throughput and the service time, as its attributes. If the user wants to ask a question about throughput and average waiting time for the banking model with input parameters mean interarrival rate being 4.0, mean service time being 6.0 and the number of tellers equal to 2, then SIMODULA checks in the BankScenario relation if this model has been run before. If it was, it retrieves the answer from relation BankScenario (values of attributes Throughput and AverageWaitingTime). Otherwise, SIMODULA launches the simulation with the input parameters retrieved from the query and the rest of them set to defaults. The authors call this approach "query driven simulations."

SIMODULA provides a good first approach to asking questions about outcomes of simulations. However, the work in [MW89] can be expanded in various ways. First of all, SIMODULA queries do not really *drive* simulations; they just *launch* them. Once SIMODULA determines that the

[1]We can draw a comparison between Query-Driven Simulations and relational query languages such as SQL. As the relational query languages are much more declarative than the earlier navigational query languages, such as DL/1 [Dat77], so is the QDS approach in comparison with SAGS.

simulation has to be run, the query does not interact with the model or influence its execution in *any* way; only at the end of the run, does the query select the statistics of interest to the user. Secondly, SIMODULA users cannot ask arbitrary *ad-hoc* queries about simulation outcomes of SIMODULA models because the queries are restricted to questions about statistics pre-defined by the modeler in the schema of the model instance relation (e.g. BankScenario relation). The major reason for this is that SIMODULA queries are not evaluated against the trace files generated by simulations but are only limited to the predefined set of statistical outcomes. Thirdly, as related to the second point, the users can ask queries only about statistical *values*. They cannot ask questions of a more general nature dealing with non-single-valued outputs, such as lists of customers who visited the bank on a certain day, or some of the questions presented at the beginning of this section (e.g. which machines have a utilization ratios of more than 90%). Finally, SIMODULA lets the user ask queries only in *one* query language (extension of SQL) against the models written in *one* simulation language (Modula [Wir85]).

In this paper we present a more extensive approach to Query-Driven Simulations by allowing SimQL queries to *drive* simulations and not just *launch* them. We also allow the user to query simulation traces in an ad-hoc manner instead of letting him/her ask a fixed set of questions on summary statistics about a single simulation run. Furthermore, we express answers in statistical terms that require more than a single simulation run to obtain the answer. Finally, we allow a loose coupling between *any* database query language and *any* simulation language as long as trace files generated by simulation programs conform to a certain standard.

In [Tuz92, Tuz93], the idea of asking queries on simulation traces was proposed, and a SimTL language was presented. SimTL consists of the simulation and the querying components. The simulation component is based on a temporal logic programming language [AM89], and the querying component is based on temporal logic [MP92]. Thus queries about simulation outcomes expressed in temporal logic are asked about simulations generated by temporal logic programs. This means that SimTL is a tightly coupled simulation and querying system, in which both components depend on the formalism of temporal logic.

In this paper, we extend the work of [Tuz92, Tuz93] by integrating an *arbitrary* temporal query language with an *arbitrary* simulation language. Therefore, unlike SimTL, where the interface between querying and simulating components is well-understood and is based on temporal logic, we have to develop a proper interface between these components in order to achieve independence between the query and simulation languages. In addition to this interface, we also describe an architecture that supports query-driven simulations.

We begin our description of the QDS system Cassandra$^+$ by informally introducing its query language component SimQL in Section 2. In Section 3, we formally define the syntax and the semantics of the language. In Section 4 we describe how QDS system integrates querying and simulation subcomponents. In Section 5, the architecture of a specific QDS system Cassandra$^+$ is described. Finally, in Section 6 we describe the implementation of the system.

2 Overview of SimQL

SimQL is a language for asking queries about outcomes of simulations. It consists of two subcomponents: the *core* query language subcomponent and the shell into which the core query language is embedded. The core query language subcomponent is the "heart" of SimQL and is used for asking temporal queries about simulation traces. We have chosen a *temporal* query language as a core query language because queries expressed in it are about process evolving over *time*. In the Query-Driven Simulation approach, we can ask queries about simulations in *any* temporal query language as long as it supports the same data model as the simulation component [2]. In this paper, we selected the temporally ungrouped historical data model [CCT93] as the underlying model for storing simulation traces and the model for the query language. Therefore, we can use any temporal query language based on this temporally ungrouped data model, e.g. we can use TQuel [Sno87], TSQL [NA88], or temporal logic calculus [TC90, CCT93].

The second subcomponent of SimQL is the *shell* into which the temporal query language is embedded. This shell provides an *interface* between the querying and simulation parts of Cassandra$^+$ that integrates the two components into one system. For example, we specify in the shell such information as the simulation model against which the query is asked, the parameters for that model, for how long simulations should be run, what answer we expect back, i.e. a full relation or just a number, and various additional information that the simulation component of Cassandra$^+$ needs in order to provide the answer to the query.

Example 1 Consider the following query:

How many parts can be finished in the next 10 hours?

It can be expressed in SimQL as:

[2] We will discuss this point further in Section 3.

Initialization:	Real-time
Type:	Event-based
Answer-Semantics:	Numeric
Core Query:	{ COUNT(Part) | **within_time** (10) Finished(Part) }
Model-Name:	Mfc-Model-4
Confidence-coefficient:	90
Error-of-estimation:	10

The core query in this example is

{ COUNT(Part) | **within_time** (10) Finished(Part) }

It is expressed in temporal logic, where **within_time** (T) P is a bounded temporal operator [Koy90, Tuz92] specifying that predicate P is sometimes true between *now* and $now + T$.

Note that this core-query is embedded in the SimQL shell that provides additional information about the meaning of the query. For example, the parameter **Model-Name** in the shell specifies the name of the simulation model, **Model-Name** = Mfc-Model-4. It tells Cassandra$^+$ that the query is asked against the model Mfc-Model-4.

The parameter **Initialization** = Real-time, specifies that simulations should be done in "real-time," i.e. they should start from some initial state of the system for the model Mfc-Model-4. Alternatively, they could be done "off-line," meaning that the initial state of the system is not specified, and simulations should be run for some time until, e.g., the steady state is reached, and only then the query should be evaluated.

The second parameter in the query. **Type** = Event-based, specifies that the trace file(s) of the simulation model Mfc-Model-4 must be stored as a historical event relation [Sno87]. In particular, the trace file of `Finished` has a *single* timestamp associated with it. If **Type** = Event-based then the simulation trace file(s) are copied into the temporal database without any conversion. Alternatively, the **Type** parameter can be "predicate-based," and this requires conversion from the event-based to a historical interval relation [Sno87]. This conversion will be described below in Section 5.6.

The value of the **Answer-Semantics** parameter in the shell is *numeric*. It specifies that the query returns back a single number (the number of finished parts in our case). Alternatively, the answer-semantics can be *non-numeric*. In this case, the query returns back a relation. We will discuss this semantics in Example 2. We have to distinguish between numeric and non-numeric semantics because the types of answers are different in these two cases as Example 2 will show.

Finally, the parameters **Error-of-estimation** and **Confidence-coefficient** specify what should

the estimation error of the answer be and with what confidence we can provide the answer [MWS90]. In our example, the user wants the estimation error to be within 10% of the mean and the confidence of the answer should be 90%.

A possible answer to this query can be:

> The average number of parts produced within the next 10 hours is 32 ± 3, and we can make this statement with confidence 90%.

In other words, the probability that the answer to the query falls between 29 and 35 parts is 90%.

□

When the user issues a SimQL query, Cassandra$^+$ determines the simulation model to which the query refers, determines how many simulation runs (N) are needed to obtain the answer within the estimates specified by the user (parameters **Error-of-estimation** and **Confidence-coefficient**)[3], runs this simulation model for N simulation runs, storing simulation traces in trace files, converts the resulting simulation trace files into the temporal database format according to the **Type** parameter, issues the temporal query against each simulation trace, and statistically analyses the answers to these queries.

We considered the first example of a SimQL query, and described some of the shell parameters. It turns out that there are other parameters in the query that were taken as *default* parameters. For example, **TIME** is one of such parameters. If not specified, it is "extracted" from the query (10 hours in our case). If it is present then we assume that the time domain in the temporal logic operators appearing in the query is restricted by this parameter, as the next example shows.

Example 2 Consider the following question that a foreman may want to ask:

> What are the parts that will always stay in Cell-1 for the next 5 hours?

This query can be expressed in SimQL as

[3] We will discuss the process of estimating N in Section 5.

Initialization:	Real-time
Type:	Predicate-based
Time:	5 hours
Answer-Semantics:	Relational
Core-query:	{Part | **always** Visits(Part, C1)}
Model-Name:	Mfc-Model-2
Parameters:	number_of_cells = 5, job_arrival_rate = 10
Confidence-coefficient:	95
Error-of-estimation:	10
Number-of-answers:	2

The core query in this example is

{Part | **always** Visits(Part, C1)}

where **always** P is a temporal necessity operator that is true now if P is always true in the future [MP92]. Note that the core query is unbounded in the sense that we need to know the values of the `Visits` predicate at all the (arbitrarily far away) points in the future to evaluate its value at present. To solve this problem, we specify the **Time** parameter in the shell. The **Time** parameter restricts the temporal domain to the bounded set of times (up to 5 hours from now), and the temporal operators are evaluated on *that* domain.

This query has additional parameters that did not appear in the previous example since default values were assumed for them in Example 1. One of these parameters is **Parameters** that specifies the parameters passed to the simulation model specified in the query. For example, `number_of_cells = 5` and `job_arrival_rate = 10` mean that these parameters are passed to the `Mfc-Model-2` model.

The **Answer-semantics** parameter in the query in this example has `relational` as its value. This means that the query returns relations as its answer. `Relational` is one example of *non-numeric* type of the **Answer-semantics** parameter, the other type being `tuple`. Both values of the non-numeric type return relations as answers to SimQL queries. However, there are some differences between these two values. We will discuss these differences when we describe the `tuple` value of the **Answer-semantics** parameter in Example 3.

The **Type** parameter has value `predicate-based`. This means that the relations in the core-query are predicates with two timestamp attributes, begin-time and end-time (unlike events that have only one timestamp attribute). For example, predicate `Visits(Part,Cell)` has two times associated with it i.e., when a part begins and ends its visit to a cell. Finally, the parameter **Number-of-answers** specifies the number of the most likely answers the user wants specified in the order of decreasing probabilities of these answers. This parameter can appear only in the

SimQL queries that have non-numeric values in the **Answer-semantics** parameter.

A possible answer to the query from this example can be

> Most likely, parts PY346, PY378, and PZ216 will always be in Cell-1 within the next five hours; the probability of this is 24% ± 2%, and we make this statement with confidence 95%. The second most likely answer is that parts PZ289 and PY378 will always be in Cell-1 within the next five hours; the probability of this is 21% ± 2%, and we make this statement with confidence 95%.

The query returns two most likely answers because the parameter **Number-of-answers** is 2 in this case. Furthermore, the answers are returned in the decreasing order of their average probability estimates.

Note that the answer to this query is different from the answer to the query in Example 1. This query returns the relation that is the most likely answer to the query and an estimation of the *probability* of that answer (e.g. probability estimate is 24% ± 2%). In contrast to this, the answer to the query with the numeric value of the **Answer-semantics** parameter returns the average estimate of the *value* of the numeric parameter and the estimation error for this value (32 ± 3 parts in that example).

□

In Example 2, we considered the relational value of the **Answer-semantics** parameter. This value directs Cassandra$^+$ to return the most likely answer(s) to the query. However, the user may sometimes want a different kind of the answer, as the following example shows.

Example 3

Consider the query

> How many days would it take to complete order number JC-243 by each of the three manufacturing plants (PL-1, PL-2, PL3)[4]?

The relational semantics would return a certain answer, e.g. { `(PL-1, 10days), (PL-2, 14days), (PL-3, 12days)` } and would assign a probability estimate for the *whole* relation, e.g. probability 26% ± 2% that the above relation is the answer. However, we may need a different answer. We may want to know probability estimates for each plant *separately*, e.g., { `(PL-1,`

[4]We assume that it should not take more than 30 days to complete the order in all 3 plants.

10days) with probability 23 ± 2%, (PL-2, 14days) with probability 34 ± 3%, (PL-3, 12days) with probability 21 ± 2% }.

To accommodate for this type of answer, we provide the *tuple* value for the **Answer-semantics** parameter, as the following SimQL query shows

Type:	Event-based
Time:	30 days
Answer-Semantics:	Tuple
Core-query:	{ Plant, Days | **sometimes** Completed(Order,Plant,Days) and Order = JC-243 and (Plant = PL-1 or Plant = PL-2 or Plant = PL-3) }
Model-Name:	Mfc-Model-6

□

There are two additional issues related to the process of interaction between SimQL queries and simulations. First, there is a *model management* issue [Bla92]. Cassandra$^+$ must store a set of simulation models against which the user can ask queries. For instance, in Example 1, the query was issued against the manufacturing model Mfc-Model-4, in Example 2 against model Mfc-Model-2 and in Example 3 against model Mfc-Model-6. Therefore, Cassandra$^+$ must store, retrieve and update various models. We will discuss this issue further in Section 5.3.

Second, different models in the modelbase can be written in different simulation languages. For example, Mfc-Model-2 can be written in MODSIM [BDMR90], Mfc-Model-4 in Simscript [Con87], and Bank-Model-12 in Simkit [Int85]. As was stated already in the introduction, one of the important advantages of Cassandra$^+$ is that it can support *any* temporal query language and *any* simulation language as long as the two agree on the data model (so that temporal queries can be asked against the corresponding traces)[5].

The next example shows that SimQL queries can be asked not only about the future but also about the past[6], and simultaneously about the past *and* the future.

Example 4 How many parts will be produced this month, assuming that now is September 15.

[5]The only convention is that the trace files generated by programs written in different simulation languages must have a certain format. The structure of this format will be discussed in Section 5.5.

[6]To simplify the presentation, we assume that the present is a special case of the past throughout the paper. The past and the present are grouped together since in both cases we know the state of the system and do not have to do simulations to answer the queries about the past *and/or* the present (just do the retrieval from the database).

Initialization:	Real-time
Type:	Event-based
Answer-Semantics:	Numeric
Time:	Combined(Past(15), Future(15))
Core-query:	{ COUNT(Part) | **within_time** (15days) Finished(Part)}
Model-Name:	Mfc-Model-1

Note that the **Time** parameter in the query indicates that the simulation model Mfc-Model-1 should be run for 15 days and then the simulation results should be *combined* with the history of relation **Finished** over the past 15 days[7]. Finally, the core query is evaluated on the combined relation that has the lifespan of 30 days.

□

The next example shows how SimQL queries can be used in the experimental design.

Example 5 How many customers will be serviced in the bank per day if we vary the number of tellers between 5 and 8?

Initialization:	Off-line(Steady-state)
Type:	Predicate-based
Answer-Semantics:	Numeric
Time:	1 day
Core-query:	{COUNT(Customer) | **within_time** (1day) Serviced(Customer,Teller)}
Model-Name:	Bank-Model-7
Model-parameter:	number_of_tellers = 5..8
Confidence-coefficient:	95
Error-of-estimation:	10

This query is called a *range* query because it gives rise to *four* individual queries, one query for *each* number of tellers (5, 6, 7, and 8) specified in the **Model-parameter** parameter. As a result of this, SimQL returns four different answers to the user, one answer per one value of parameter.

Also note that this is an *off-line* query. This means that simulations are run initially until Bank-Model-7 reaches a steady state [LK91]. Only after that, simulations will be run for one day of simulated time and traces will be generated starting from that time.

□

In this section we informally introduced the language SimQL through a series of examples. In the next section, we formally define the syntax and the semantics of SimQL.

[7]The historical data can be obtained by gathering the transactional real-time data about all the events and activities happening in the enterprise, as is done in [FDJG+92], and then processing this data and storing it in the historical relational database format.

3 Formal Description of SimQL

We begin the formal description of SimQL with the specification of its syntax. In Section 3.2, we describe the semantics of the language.

3.1 Syntax of SimQL queries

The BNF grammar for SimQL is presented in Figure 1. We assume that entities `value`, `number`, `INTEGER`, `REAL`, and `STRING` are terminal symbols, and we do not decompose them any further. We explained most of the SimQL's features informally in Section 2 already. Therefore, we will briefly describe some of the most important parts of SimQL that received less attention in Section 2.

We start with the shell parameters. First each parameter in the shell has a default value. The default values for the parameters in Figure 1 are italicized. Second, the core query can be expressed in *any* temporally ungrouped historical query language [CCT93]. In Section 2, examples were presented in temporal logic calculus that was described in [Tuz92]. However, we could have done it in any other temporally ungrouped query language [CCT93], such as TQuel [Sno87] or TSQL [NA88][8]. Third, as Figure 1 shows, the off-line value of the **Initialization** parameter can be *Steady-state*, *Zero*, or *Time = number*. This means that the initial state of the system from which simulations begin their executions is obtained as follows. If the value is "Steady-state" then the initial state is obtained by running simulations until the steady state is reached. This steady state serves as the initial state for simulations. This is the standard approach often taken in simulations [LK91]. If the value is "Zero" then the initial state is obtained by setting all the values of all the parameters to zero. For example, in a banking application, we can start simulations by assuming that the bank just opened, and its branches have no customers yet. If the value is "Time = t" then the initial state is obtained by running simulations for time t and taking that state as the initial state. Fourth, the **Time** parameter has values "future," "past," and "combined." For the "future" and "combined" values of the **Time** parameter, simulations are launched as described in Section 2. If the parameter has the value "past" then the query is evaluated on the *past* history of the database, and most of the other parameters are not applicable in this case. The only parameters that can be used in this case besides the **Core-query** parameter are **Time** and **Model-name**.

We next describe the syntax of the core queries from Figure 1. As we said already, *any* temporally ungrouped query language can be used as a core query language. Since examples of

[8] Although a core query can be expressed in any of these query languages, it is an entirely different matter how Cassandra$^+$ can accommodate different DBMSes that support these query languages. We will address this question is Section 6.

queries :	[initialization][type][answer-semantics] [time] simple-query [model-name] [model-parameters] [error-of-estimation] [confidence-coefficient] [number-of-answers]
initialization :	**Initialization**: Off-line (off-line-type) | **Initialization:** *Real-time*
off-line-type :	*Steady-state* | Zero | Time = number
type :	**Type**: *Event-based* | **Type**: Predicate-based
answer-semantics :	**Answer-Semantics**: Numeric | **Answer-Semantics**: non-numeric
non-numeric :	Tuple | *Relational*
confidence-coefficient :	**Confidence-Coefficient**: number
error-of-estimation :	**Error-of-estimation**: number
number-of-answers :	**Number-of-answers**: number
time :	**Time**: Past(range) | **Time**: [*Future*] (range) | **Time**: Combined(Past(range), Future(range))
simple query:	**Core-Query**: < Expressed in *any* temporal query language >
model-name :	STRING
model-parameters :	**Model-parameter:** name = range {, name = range}
value :	INTEGER | REAL | STRING
number :	INTEGER | REAL
range :	INTEGER..INTEGER STEP INTEGER | INTEGER

Figure 1: BNF Grammar of SimQL

core queries in Section 2 were described in temporal logic calculus, we will use this calculus as the core query language in the paper. Furthermore, it also does not matter what kind of temporal logic is used in this calculus, since various types of temporal logics can be incorporated into Cassandra$^+$.

Temporal logic is obtained from first-order logic by adding various temporal operators to it. The most commonly used temporal operators are the future operators **sometimes_in_the_future** ($\diamond$), **always_in_the_future** ($\Box$), **next** ($\circ$), **until** and their past "mirror" images **sometimes_in_the_past** ($\blacklozenge$), **always_in_the_past** ($\blacksquare$), **previous** ($\bullet$), and **since**[9]. Besides these eight standard operators, other temporal operators can be defined, such as **before**, **after**, **while** [Kro87], and bounded necessity, **for_time (T)** ($\Box_T$), and possibility, **within_time (T)** ($\diamond_T$), operators [Tuz92]. The meaning of future operators is defined in Fig. 2. The meaning of past "mirror" images of these operators is defined similarly to the future operators except time is referenced only in the past. Kroger [Kro87] shows how temporal operators **before**, **after**, and **while** can be expressed in terms of the operators **until** and **since** [Kro87]. Furthermore, it easily follows from the expressive completeness of the temporal logic *US* [Kam68] for the discrete or continuous model of time, that the operators of bounded necessity and possibility can also be expressed in terms of the **until**, **since**, **next**, and **previous** operators for these models of time.

The following example illustrates the use of temporal logic.

Example 6 The statement

> If a person is promoted from the Associate to the Full Professor he/she cannot be demoted back in the future to the rank of Associate Professor at the same school.

can be expressed in temporal logic as

[9]Note that the operators $\diamond$ and $\Box$ can be derived from $\circ$ and **until**; and $\blacklozenge$ and $\blacksquare$ from $\bullet$ and **since** [MP92]. Also note that if we consider operators **next** and **previous** then time must be discrete. Alternatively, we can remove these operators and consider the continuous model of time.

$\Box A$	is true now if A is always true in the future (**necessity**)
$\diamond A$	is true now if A is sometimes true in the future (**possibility**)
$\circ A$	is true now if A is true at the next moment of time (**next**)
$\Box_T A$	is true now if A is always true within the next T time moments (**bounded necessity**)
$\diamond_T A$	is true now if A is sometimes true within the next T time moments (**bounded possibility**)

Figure 2: Temporal Logic Operators

$\blacklozenge$ $EMPLOY(Name, School, Associate\_Prof) \wedge EMPLOY(Name, School, Full\_Prof) \rightarrow$
$\Box\neg EMPLOY(Name, School, Associate\_Prof)$

or using a different syntax as

IF sometimes_in_the_past *EMPLOYEE(Name, School, Associate_Prof)* and *EMPLOYEE(Name, School, Full_Prof)* **THEN always_in_the_future** not *EMPLOYEE(Name, School, Associate_Prof)*

□

A *temporal logic query* is an expression of the form

$$\{X_1 \ldots X_n \mid \phi(X_1, \ldots, X_n)\}$$

where ϕ is a predicate temporal logic formula in the language just described containing free variables $X_1 \ldots X_n$. The answer to this query consists of all the tuples $X_1 \ldots X_n$ that make the formula $\phi(X_1, \ldots, X_n)$ true. Core queries in Examples 1 - 5 provide examples of temporal logic queries.

3.2 Semantics of SimQL queries

In this section we will explain what an "answer" to a SimQL query means. Recall that in Examples 1, 2 and 3 the answer to the query is defined in statistical terms, and that it is necessary to do several simulation runs to produce that answer. In order to define semantics of queries for multiple simulation runs, we first have to explain what the answer to a query means for a *single* simulation run.

3.2.1 Semantics of a Core Query for a Single Simulation Run

A SimQL query on a single simulation run is evaluated as follows. As will be explained in Section 5.5, a single simulation run generates a trace file that contains occurrences of the events traced by the simulation program. After that, this simulation trace file is converted into a temporal database containing one temporal relation per each event being traced. Furthermore, all the temporal predicates referred to in the core query must appear in this temporal database. Then the core query is evaluated on this temporal database according to the semantics of the language in which the core query is expressed. In particular, if it is expressed in temporal calculus then it is evaluated according to the semantics of that calculus [TC90, CCT93]. This semantics assumes that the temporal database forms a *temporal structure* [Kro87] for the temporal logic formula, and then

model checking is done against this temporal structure, i.e. the truth value of the formula at the present moment of time is evaluated in terms of this temporal structure.

For instance, consider the query from Example 1. The execution of simulation model Mfc-Model-4 specified in that query results in the generation of the trace file containing all the occurrences of the event `Finished(Part)` that happen within 10 hours of simulated time from now. For example, some events in this trace file may be { `Finished(P3, 10:23)`, `Finished(P6, 10:47)`, `Finished(P8, 11:13)` }. Since the type of the query is event-based, events are moved from the simulation trace file into a temporal database without any additional conversion. Let `FINISHED` be the temporal relation storing these events. Then the core query

{ COUNT(Part) | **within_time** (10 hours) Finished(Part) }

is evaluated on the temporal relation `FINISHED`, and the result is a single number equal to the number of finished parts.

We want to point out one caveat in this evaluation process within the context of query driven simulations. When the user specifies the **TIME** parameter in the query shell, it means that the *temporal domain* for the core query is restricted to the times specified by the user. For example, if the parameter is **TIME** = 10hours, and it is 11 a.m. on 10/12/93 now, then the temporal domain is from 11 a.m. 10/12/93 to 9 p.m. 10/12/93 for the core query in this example, and all the temporal operators in the query are evaluated for *this* domain.

3.2.2 Semantics of SimQL Queries for Multiple Simulation Runs

Once we know the answer to a SimQL query for a single simulation run, we can explain the semantics of the query for multiple simulation runs. However, as we mentioned before, we have to distinguish between the two cases when the core query returns a *numeric* and when it returns a *non-numeric* answer because the answers for multiple runs are quite different in these two cases.

Numeric Semantics of Answers. In this case, a query returns a number per a single simulation run, and thus multiple simulation runs generate sets of numbers, one number per run. Also, some of these numbers may be repeated in the set. For instance, assume we do five simulation runs for the query from Example 1 and assume we get the answers { 18, 20, 19, 18, 19 } for these runs. To determine the answer to a "numeric" query, we assume that this answer is normally distributed. Then we estimate the mean and variance of this normal distribution from the sample of answers to individual runs and determine the *confidence interval* [MWS90] for the average answer based on the confidence coefficient specified in the query. If this confidence interval has an error of estimations

not greater than the one specified by the user in the query, we stop the simulations. If not, we increase the number of simulations to be run in order to get the confidence interval within the limits and run that many simulations again.

Therefore, the semantics of a SimQL query for the *numeric* type of an answer is defined by an estimate of the average value of the answer, by the error of estimation and the confidence coefficient specified in the query. For example, if the mean value of the number of parts that will be finished within the next 10 hours is 20, the estimation error is 15%, and the probability that this number is between 17 and 23 is 95% then the answer that Cassandra$^+$ returns to the user who asked this query is

> The average number of parts produced within the next 10 hours is 20 $\pm$ 3, and we can make this statement with confidence 95%.

Non-Numeric Semantics of Answers. In this case, the query returns a relation, and not a single number[10]. Since we consider relations instead of numbers, we cannot make statements about averages for these relations. Instead, we want to determine which answers are the *most likely* ones. However, there are two ways of doing this. The first alternative is to determine which answer, as a *relation*, is the most likely one. This alternative can be selected by specifying **Answer-semantics** = `relational` in the query. The other choice is to determine which tuples in the answer are the most likely ones. This alternative can be selected by specifying **Answer-semantics** = `tuple`. We describe relational and tuple semantics of answers now, assuming that the parameter **Number-of-answers** in the query is equal to N.

If *relational semantics* is selected in the query then we compute frequencies for each relation returned as an answer for all the simulation runs. In this case we treat each answer as an alternative outcome of simulations and want to know the most probable outcomes. For instance, assume that we made five simulation runs for the query from Example 2 "*what are the parts that will always stay in Cell-1 for the next 5 hours,*" and assume we get the frequencies for each of the resulting answers as shown in Figure 3.

As we increase the number of simulation runs, the distribution of the estimate of the frequencies of each simulation outcome converges to a normal distribution[MWS90]. Then our goal is to estimate N largest frequencies based on the parameters **error-of-estimation** and **confidence coefficient** specified in the query.

[10]Of course, the relation can also consist of a single number in the degenerate case. For example, we could specify **Answer-Semantics** = `Relational` for the query in Example 1. However, if we did so, we could not talk about an average number of parts produced in 10 hours.

ANSWER	*FREQUENCY*
{PY346, PY378, PZ216}	0.4
{PY346, PZ216}	0.2
{PU629, PY378, PZ216}	0.4

Figure 3: Frequencies of Different Answers for the Query From Example 2.

The semantics of a SimQL query for the *relational* type of an answer is defined by N relations having N ***largest*** frequencies in this distribution (based on the estimation of their means), i.e. the query returns the first N ***most likely*** answers. If $N = 1$ then the query returns a most likely answer. If there is more than one most likely answer then either all of them can be returned, or one of them selected at random. In our example, either both most likely answers, { PY346, PY378, PZ216 } and {PU629, PY378, PZ216}, having frequency 0.4, or one of them chosen at random is returned if relational semantics is selected. We assume that $N = 1$ as the *default* value for the relational semantics of answers.

If *tuple semantics* is selected in the query, then we compute frequency of occurrence of *each tuple* in the set of answers. In other words, we want to know the chance of each tuple belonging to the answer. For example, if we made K simulation runs, and the tuple $(a_1, \ldots, a_n)$ occurred in answers for k runs, then the frequency of $(a_1, \ldots, a_n)$ is $\frac{k}{K}$. Then the semantics of a SimQL query for the *tuple* type of an answer is defined by the first N tuples having the highest frequencies. Furthermore the default value for the parameter **Number-of-answers** is *All*, i.e. the user wants to know frequencies of occurrence of all of the tuples in the answers. For example, if five simulation runs produce answers as presented in Figure 3, then the answer to the query from Example 2 based on the tuple semantics is shown in Figure 4 for **Number-of-answers** = *All*. In other words, the tuple semantics for this query specifies the chance various parts will always stay in Cell-1 for the next 5 hours.

In summary, we showed how the same non-numeric query can have two different answers depending on whether the semantics is relational or tuple-based. The two examples presented above show that both semantics may be needed in practice, and that it is up to the user to select the semantics he or she wants.

Once we know the semantics of answers for SimQL queries, our next task is to determine the number of simulation runs necessary to answer a query. We describe how this number is obtained in Section 5 when we present the query evaluation strategy of Cassandra$^+$.

TUPLE	FREQUENCY
PZ216	100%
PY378	80%
PY346	60%
PU629	20%

Figure 4: A Sample Tuple Semantics Answer

4 Query Driven Simulation

As explained in the introduction, *Query Driven Simulations (QDS)* is an approach to simulations in which the user first asks queries about outcomes of simulations and then, depending on the query being asked, appropriate simulations are launched, and events necessary to answer the query are recorded in the trace files. After simulations are finished, the query expressed by the user is evaluated on the trace files(s) being generated by simulations.

In Sections 1 and 2 we showed the kind of questions that people ask about future outcomes of business process in organizations. Currently, to answer such questions, users rely on systems developed by the information systems departments that support a fixed set of questions, one program handling only a few questions (only a single question in the worst-case scenario). This "program-per-question" approach can be satisfactory when users have few questions. However in other situations, decision makers might have very many questions of interest, and it makes a-program-per-question approach either impossible or very costly.

The situation becomes even more difficult when an organization has multiple simulation models since a program has to be written per each question per each model in the worst-case scenario. For example, if decision makers want to ask 500 different questions against 20 different manufacturing models then this means that one may have to write 10,000 programs in the worst case that handle these questions.

This discussion suggests the types of applications where QDS systems, and Cassandra$^+$ in particular, are most useful. These applications can be measured in terms of two dimensions: how many models are there in the modelbase and how many questions different users of a QDS system want to ask about these models.

The best types of applications are those where users want to ask many queries about various types of models since in this case the alternative program-per-question approach is the most ex-

		Models: Few	Models: Many
Queries	**Few**	Fair	Good
	Many	Good	Excellent

Table 1: Applicability of Query-Driven Simulations to Different Types of Applications.

pensive. The worst application is when there are few models and the users want to ask only few questions about simulation results since in this case the program-per-question approach is feasible. This discussion leads to the following characterization of the QDS applications presented in Table 1.

5 Architecture of Cassandra$^+$

The architecture of the Cassandra$^+$ system that supports query-driven simulations is described in Figure 5. We describe how it works from the time the user submits a query to the Cassandra$^+$ system until it returns the final answer back to the user.

When the user submits a query to Cassandra$^+$ it is passed to the interpreter. The interpreter parses the query and checks for syntactic errors. If no errors occur, the interpreter then passes the query to the controller. The controller determines the query type, i.e. past, future, or combined. Based on the query type, it decides if simulations have to be run in case of future and combined queries, or if a simple database retrieval has to be made in case the query is about the past. The controller determines the name of the model for the query based on the **Model-name** parameter and checks the modelbase, containing the collection of various simulation models, to see if this model exists. If the model exists and the query is about the past, the controller accesses the past history for the model and evaluates the query against its past history.

If the query is about the future the default simulation parameters are retrieved for the model from the modelbase. Some of these default parameters are over-written by the parameters specified by the user in the **Parameters** clause of the SimQL query. If the query is "combined" the two processing methods (about the past and future) are combined.

In order to explain how Cassandra$^+$ processes a query for a given simulation model, we have to explain the following points. First, we have to explain how Cassandra$^+$ determines the number

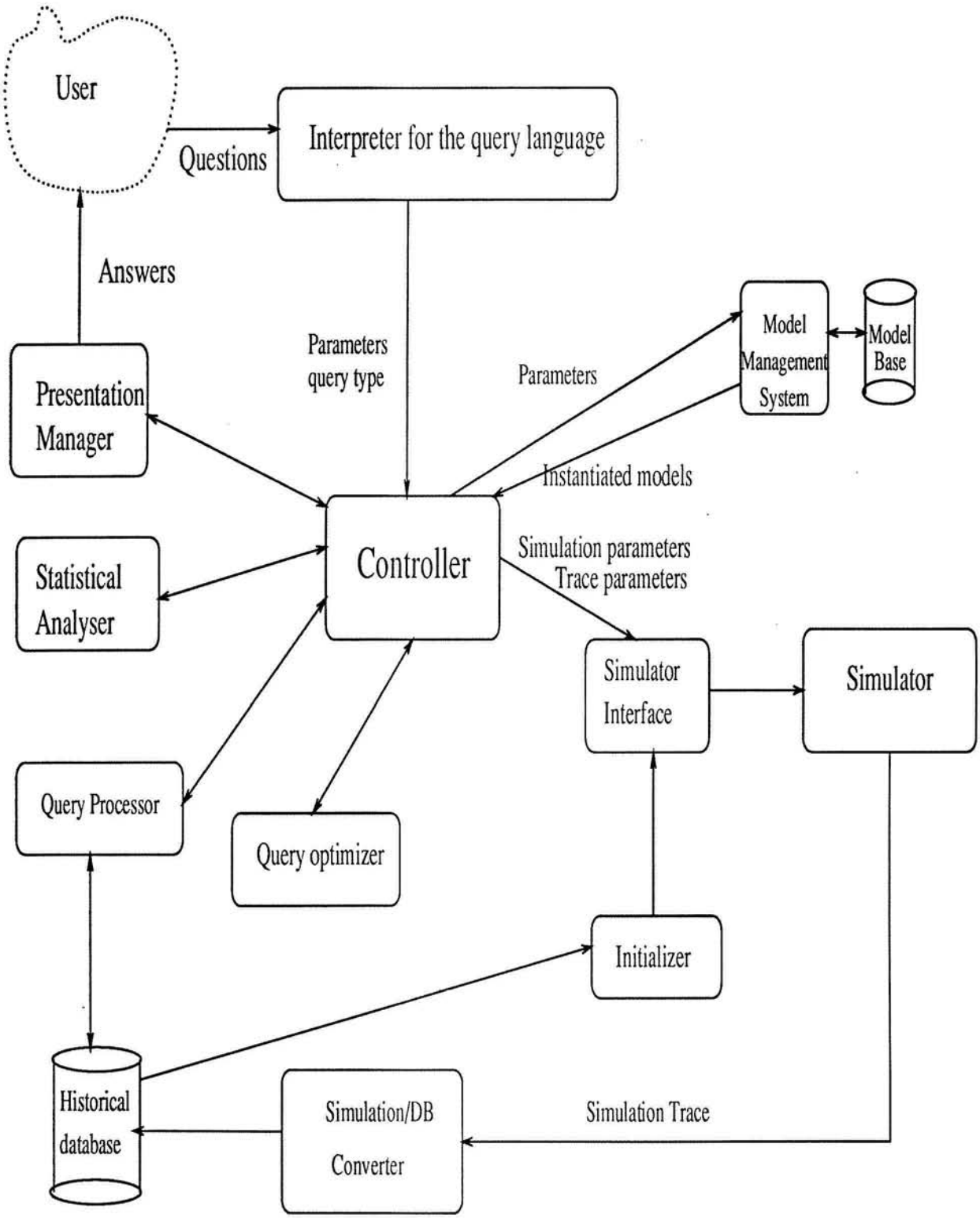

Figure 5: Architecture of Cassandra+

of simulation runs. Secondly, we have to explain how Cassandra$^+$ processes one individual run. Finally, we have to explain how answers to queries for individual runs are combined together and are statistically analyzed. We first describe the second part, i.e. how Cassandra$^+$ answers a query for a single simulation run.

As a first step towards answering a query on a single simulation run, the simulation model specified in the query is instantiated with the input parameters. Some of these parameters are taken as default values from the modelbase and others from the query itself. Also, the simulation model is initialized with the initialization information as will be described in Section 5.4. Furthermore, the query passes the information to the simulation model about the events the model should trace. We will describe this further in Section 5.10.

Once the simulation model is initialized, the actual simulation is launched. During the simulation process, the simulation program writes various events that the query asked it to trace into the *trace* files, one file per one event. For example as we pointed out already in Section 3.2, various occurrences of event FINISHED for the query from Example 1 are recorded into the trace file FINISHED that may have the events { `Finished(P3, 10:23)`, `Finished(P6, 10:47)`, `Finished(P8, 11:13)` } recorded in it.

After that, the trace files containing events are converted into the temporal database relations. If **Type** parameter in the query is "event-based" then the conversion process is simple and is done on a record-by-record basis: one event in the trace file generates the corresponding record in the temporal relation. If **Type** parameter in the query is "predicate-based" then we have to convert events into predicates using conversion routines stored in the modelbase. After the conversion process, the trace files are stored in the temporally ungrouped historical database format [CCT93]. If the query is of the type "combined" then Cassandra$^+$ merges the past history of predicates and/or events with the simulation traces converted to the database form. After that, the core query is evaluated on the temporal database. We will describe the details of this process below. This completes the description of how a SimQL query is evaluated on a single simulation run.

Another important task for the controller is to determine the number of simulations to be run if the query is about the future. To do this we start with an initial set of simulation runs, run them and see if the answer satisfies the estimation error and confidence level specified in the query. If it does not, the number of runs is increased, and the process continues until the answer satisfies the constraints specified in the query. For example, assume that we do five simulation runs for the query from Example 1 and we get the following answers {18, 22, 19, 25, 17 } for these runs. The mean value of this sample is 20.2 and the variance is 10.7. We then calculate

the 95% confidence interval for the parameter which in this case is 20.2 ± 5.38. If the user wants the estimation error of the answer to be within 10% of the mean, then we have not satisfied that constraint and we have to increase the number of simulation runs N and continue the simulations. For example if the user wants the answer to be within 2 parts, we increase the number of simulation runs to 11 using the calculations described in [TB93]. Assume that we get the following answers {18, 19, 18, 20, 19, 18, 19, 17, 19, 20, 19 }. We again compute the confidence interval which in this case turns out to be 18.72 ± 1.48. This value is again outside the bounds that the user specified (2 parts). Therefore, we again compute the new sample size and continue this process until the confidence interval falls within the bounds specified in the query by the **Error-of-estimation** and **Confidence-coefficient** parameters. Details of this algorithm can be found in [TB93].

This completes our description of the query processing method used in Cassandra$^+$. In the next section we describe components of Cassandra$^+$ architecture, as presented in Figure 5, i.e., interpreter, controller, modelbase, simulation interface, simulator, simulation/database converter, query processor, statistical analyzer, presentation manager, and query optimization modules.

5.1 Interpreter

The queries that the user enters are sent to the interpreter which parses the query to determine query type, model name and model parameters, simulation time, the answer semantics, etc., and passes this information to the controller. The core-query remains *untouched* by the interpreter which means that it is passed to the controller and later to the query processor as an unparsed string without any modifications. As Example 5 and Figure 1 show, SimQL supports range queries. Therefore, the interpreter also determines the ranges in model parameters specified in the query and passes this information to the controller. For instance, when the interpreter parses the query in Example 5, it will tell the controller that Bank-Model-7 has to be run with the number of tellers ranging from 5 to 8.

5.2 Controller

The controller manages the query processing activities by delegating various processing tasks to different modules, receiving results back from these modules and interpreting these results. Broadly, the functions of the controller can be classified as follows:

- Determination of the query type
- Model manipulation

Number of tellers	Sim_st	Sim_fin	Number_of_runs	Trace_file
5	0	8	20	trace1
6	0	8	20	trace2
7	0	8	20	trace3
8	0	8	20	trace4

Figure 6: Instances in the working-modelbase for Example 5.

- Determination of the number of simulation runs

We describe each of these functions now.

5.2.1 Determination of the Query Type

The first task of the controller is the determination of the query type, i.e. whether the query is about the past, the future, or is combined, and setting up the query processing strategy for that query type. As Figure 1 shows, query type can be past, future, or combined.

Past Query Type. If the query is about the past, the controller initiates a simple historical database retrieval from the past history of activities for the model. The past history of the model is obtained by recording all the activities of the model in a temporal ungrouped historical database [CCT93]. For example, in the LMS system [FDJG+92], all the transactional data from a manufacturing shop floor is captured in real time and is recorded in a database for subsequent retrieval.

As an example of the past query, consider the question "*how many jobs of various types were made in the last 10 days based on the model Mfc-Model-4.*" To answer this query the controller determines from the modelbase for model Mfc-Model-4 where the data about the past history of the model is stored in historical relations, whether this data is sufficient to answer the query, and if it is sufficient, it issues the (historical) query against these relations.

Future Query Type. In this case, the controller launches simulations. However before doing it, the controller determines if the query is a range query (i.e. has several sets of parameters for the simulation model). If it is a range query, the controller creates a *working modelbase* that contains one entry for each set of parameters. For example, when the query from Example 5 is evaluated, the resulting working modelbase is shown in Figure 6. The parameters that change from one instance of the model to the next are the number of tellers shown in the first column of the table and the name of the trace file shown in the last column.

After that, a *separate* instance of the query is issued for each entry in the working modelbase table. Each request generates a separate answer which is shown to the user. For example, the first instance in the working modelbase (with 5 tellers) may return the answer:

> The average number of customers serviced within the next 8 hours is 150 ± 10, and we can make this statement with confidence 90%.

The second instance in the working modelbase (with 6 tellers) may return the answer:

> The average number of customers serviced within the next 8 hours is 226 ± 12, and we can make this statement with confidence 90%.

Combined Query Type When the query is about both the past and the future, the controller runs simulations for the future part of the query as it does for the pure future type. Once the simulation trace is converted into the temporal database format, it is merged with the past history of the model that is obtained as for the case of pure past queries. After the "past" is merged with the "future," the core query is asked on the merged temporal database.

5.2.2 Model Manipulation

This part of the controller is responsible for selecting the model from the modelbase specified in the query. Furthermore, the controller retrieves all the default parameters for that model from the modelbase. If the query is a range query, then the initialization of the model with the default parameters should be done for *each* entry in the working modelbase as shown in Figure 6. Finally, the controller overrides the default values of simulation model parameters with the values specified by the user in the query.

5.2.3 Determination of the Number of Simulation Runs

This part of the controller determines how many simulation runs should be performed in order to bring the estimation error for the answer within the bounds specified in the query. We explained how to do this at the beginning of Section 5.

5.3 Modelbase

The modelbase contains information about the simulation models that the user can query. It is a central repository of all the information about all the models used in an organization. The

modelbase is needed in order to instantiate and run simulation models and to convert their outputs into historical relational database formats. In Cassandra$^+$, we store the modelbase as a *set* of relational tables since the information we need in the modelbase is not normalized (for example, we store the descriptions of all the types of events a model generates), and thus it is better not to place this information in a single table. However, one table in the modelbase is the main table having model name as its key. Some of the fields in this table are listed below:

- *Model name*, that serves as a key. For instance, Mfc-Model-4, Banking-Model-6 are examples of model names.
- *Target simulation language*: the language in which the simulation model is written, e.g. Modsim, Simscript, etc.
- *Name of the simulation program*. This field contains the *name* of the object module for the simulation model, as stored in the secondary storage. When the actual simulation is ready to be run, this object module is dynamically linked to Cassandra$^+$ module using the name of the module stored in this field.
- *Default simulation parameters*: parameters that are used in the model. These parameters are taken as defaults. They can be over-written by the parameters that the user specified in the query.
- *Events traced by the model*: this field contains the name of the relation that describes all the events traced by the simulation model.
- *Event-to-predicate conversion programs*: this field contains the name of the relation that contains the names of the programs that build various predicates from the events described in the previous entry.
- *Past information*: the field contains the name of relation that stores past information about events and predicates pertinent to the simulation model.
- *Optimization flag*: this is a boolean field specifying if queries on this simulation model can be optimized. In order for a query to be optimized on a simulation model, the model should have its PRINT statements written according to a certain convention so that the query could pass the optimization information to the simulation model.

5.4 Simulation interface

Once we have determined the simulation model to be run and the parameters we have to use to run the model, an instance of the model to be executed is created in the working modelbase (Figure 6). The simulation interface is responsible for creating a run-time version of every instance of the simulation model created for the query in the working modelbase. The run-time version contains

- the *initialization* file containing the data about the initial state of the system to be simulated
- the *simulation input* file containing the simulation model parameters, simulation time taken from the query, and the number of simulation runs
- the *executable simulation program* containing the simulation model.

Therefore, the simulation interface generates the initialization and simulation input files and prepares the simulation program for the execution.

The way the initialization file is created depends on the value of the **Initialization** parameter in the SimQL query. If the initialization parameter is set to the "real time" value, then the initial state of the simulation model is set to the current state of the physical model being simulated. This is achieved by retrieving the records from the historical database corresponding to the simulated model with the time attribute equal to *now*. Then these values are passed to the simulation model as the values of the initial state of that model.

Alternatively, the initialization parameter can have the type "off-line." This means that the initial state from which actual simulations are run is not based on any actual state of the physical system being modeled but is obtained from some "hypothetical" state. This hypothetical off-line state can be obtained in the following three ways depending on the value that the "off-line" type of the **Initialization** parameter takes (see Figure 1). If it takes the value "steady-state," then simulations should be run until the steady state is reached, and then this state serves as the initial state for simulations. This is the most popular approach taken in classical simulations [LK91]. If the value of the **Initialization** parameter is "zero," then the initial state for the simulation model is specified by setting all of its initial variables to zeroes. This situation arises in a banking application, where at the beginning of each day we assume that there are no customers in the bank and no tasks pending. Finally, the last case is when the value of the **Initialization** parameter is "Time = t." This means that the simulation model must be simulated for time t starting from the zero state, and the state of the model at time t should be taken as the initial value of the simulation model.

Besides creating the run-time version of the simulation model, the simulation interface also links dynamically the object module containing the simulation program to the Cassandra$^+$ module.

5.5 Simulator

The simulator module is responsible for the actual execution of the simulation program. As we mentioned in Section 2, simulation models can be written in *any* simulation language, such as SIMAN [PSS90], SIMSCRIPT [Con87], MODSIM [BDMR90]. GPSS [IBM70], etc. The only requirement for the language is that it must be a compiled language, so that the names of object modules of programs written in this language can be stored in the modelbase as described in Section 5.3.

Another requirement for the simulation program is that it should produce traces of events of the form

$$EVENT(parm_1, \ldots, parm_n, time)$$

where $EVENT$ is the name of the event being traced. $parm_1, \ldots, parm_n$ are parameters of the event, and $time$ is the time when the event occurred. For example, the event ARRIVED(part3, cell4, 10:45am) says that the part "part3" arrived at the cell "cell4" at time 10:45 a.m. Furthermore, there should be an agreement across different simulation models and languages on the exact format of the trace files of events so that different simulation models be compatible in their outputs.

If the optimization flag in the modelbase for a simulation model is set off, then the query does not pass any optimization information to the model. This means that the simulation model does not know which events the query needs back from the simulation program, and therefore, it must record *all* its events in the trace file(s).

If the optimization flag in the modelbase is set on for a simulation model, this means that the query passes information about the events that it wants the model to record. In this case, the print statements in the simulation model must have a special form described in Section 5.10 and in [TB93].

5.6 Simulation/database converter

This module converts the simulation trace files generated by the simulator into historical relations. Since we consider temporal databases and query languages based on the relational data model, this means that we have to convert simulation trace files into the temporally ungrouped historical relations [CCT93].

Depending on the value of the **Type** parameter in the query, we either have to convert events

into historical event relations [Sno87] (if the the **Type** parameter is event-based), or convert events into historical interval relations [Sno87] according to some conversion procedures (if the **Type** parameter is predicate-based).

Conversion of events into the historical event relations is a straightforward procedure. Since the trace files are stored in the format described in Section 5.5, the conversion procedure consists of a single COPY command that converts the plain comma-delimited ASCII file into a relational database file[11].

If the **Type** parameter specified by the user is "predicate-based," then we do the actual conversion from events to interval based historical relations. This is achieved by executing the events-to-predicates conversion programs supplied by the systems administrator. For instance, consider the query from Example 2. This query deals with the temporal *predicate* `Visits` that can be represented in the interval-based temporal relational model as `Visits(Part,Cell,From-Time,To-Time)`. Furthermore, assume that the model Mfc-Model-2 has trace files for the two events `Arrives(Part,Cell,Time)` and `Departs(Part,Cell,Time)`. To construct the predicate `Visits`, the systems administrator writes the procedure that computes the values of attributes `From-Time` and `To-Time` in `Visits` from the events `Arrives` and `Departs`.

5.7 Query processor

As we stated in Section 5.1, the core query is separated from the SimQL query shell by the Cassandra$^+$'s interpreter and is kept unparsed.

The query processor evaluates this core query on the simulation traces converted into the temporal database format if the query is strictly about the future. This evaluation is done according to the query processing algorithms for the core query language and the temporal database being used. For example, if the database is Ingres and the query language is embedded SQL with timestamps, then the core query expressed in SQL is passed to Ingres and is evaluated by Ingres on the trace files that simulation/database converter generates. If the query is combined (is about the future and the past) then the future part of the database is merged with the past, and the query is evaluated on the merged database.

The result of this evaluation is either a number if the answer-semantics parameter is "numeric," or a relation if the answer-semantics parameter in the query is "non-numeric."

[11] In our implementation of Cassandra$^+$, we used Ingres [Ing89] as the relational database and its COPY TABLE command [Ing89] to do this.

5.8 Statistical Analyzer

Statistical analyzer generates the answer to an SimQL query, that is presented to the user, in the form described in Section 3.2. This answer is computed in statistical terms from the set of answers for individual simulation runs, as is also described in Section 3.2.

As we pointed out in Section 3.2. the answers and the statistical analysis techniques differ depending on the value of the **Answer-semantics** parameter. For the "numeric" parameter, the answer is in terms of the average answer of the numeric value, error of estimation and confidence interval for the estimation. For the "non-numeric" parameter, the answer is in terms of the most likely answer, its probability, error of estimation of this probability and the confidence of the estimation. The process of obtaining these estimates was described in Section 3.2.

5.9 Presentation Manager

This subsystem is responsible for converting the answers to the queries as determined by the query processor into outputs for the user. The outputs can be of various forms such as tables, charts, or graphs. For instance, the answer to the query from Example 3, i.e., "*how many days will it take to complete order JC-243 by each of the three manufacturing plants (PL-1, PL-2, PL-3)*" can be displayed to the user in one of the three forms presented in Figure 7. As in Example 3, we assume that the answer-semantics of the query is of type "tuple."

5.10 Query Optimizer

The query optimizer finds more efficient ways to process SimQL queries. However, unlike query optimizers in database management systems, it concentrates not on the query itself but on how to record only the events that are *necessary* for answering the query. In particular, it determines

- which events for a given SimQL query should be traced
- which event *instances* in these trace files should be recorded.

The query optimizer obtains this information from the query and provides appropriate directives to the simulation module. The details of this procedure, as well as other optimization issues, are described in [TB93].

Since the optimizer tells the simulator which events it should trace depending on the query, the trace files for the same simulation model can differ from each other for different queries. Therefore,

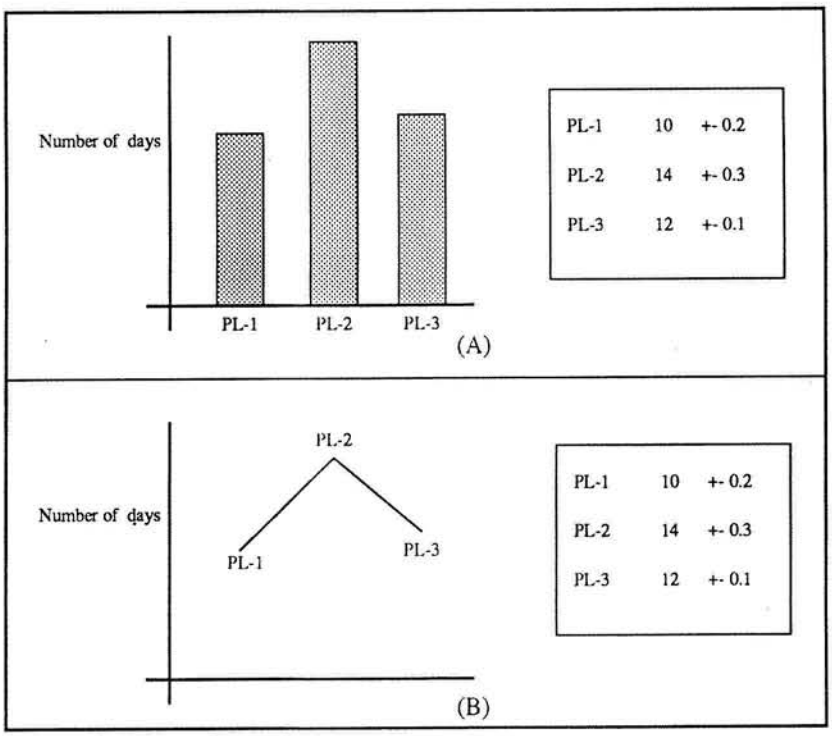

Figure 7: An Example of an Answer displayed by Cassandra$^+$

the query in our approach selects the simulation model, and the query optimizer tells the model what it should trace. It is for this reason that we call this approach Query-*Driven* Simulations.

The optimizer imposes certain constraints on how events are recorded in a simulation model. These constraints are described in [TB93]. This means, among other things, that the simulation model has to be "prepared" for optimization. To distinguish prepared from unprepared models in the modelbase, we have the "optimization flag" as one of the fields in the modelbase. If this flag is set on, this means that the model is prepared for the optimization (its PRINT statements satisfy a certain convention); otherwise, the model is unprepared.

6 Implementation of Cassandra$^+$

The architecture of Cassandra$^+$ described in this paper was implemented in C under UNIX. Since there are no commercially available temporal databases when the system was developed, we selected Ingres [Ing89, Sto86] as the database that stores historical and simulated data. We simulated a temporal database in Ingres by adding two timestamp columns to the interval based relation and a single timestamp column to the event-based relations. For example, the interval based relation `Visits` has two additional time columns and therefore

has the schema `Visits(Part,Cell,From-Time,To-Time)`. Similarly, the event-based relation `Arrives(Part,Cell,Time)` has only one temporal attribute that specifies when a part arrives at a cell.

Instead of using temporal logic calculus for the core queries, we use SQL since SQL is one of the two query languages that comes with Ingres (the other one being Quel). However, we allow SQL queries to explicitly reference time. This makes temporal queries look "ugly," but still they can express a wide range of queries.

For example, the query from Example 1 *"how many parts will be finished in the next 10 hours?"* can be expressed in SQL as:

Initialization:	Real-time
Type:	Event-based
Time:	10 hours
Answer-Semantics:	Numeric
Core-query:	
SELECT	Count(Part)
FROM	Finished
WHERE	$NOW < Finish_Time .AND. Finish_Time < $NOW + 10;
Model-Name:	Mfc-Model-4

As we said already, the modelbase can store simulation models written in various simulation languages, such as Modsim [BDMR90], Simscript [Con87], Siman [PSS90]. In fact, these programs are stored as object modules in the modelbase, and are dynamically linked to the Cassandra$^+$ module when queries are asked against these models.

We store modelbase as an Ingres database because the modelbase can be quite large and because building and maintaining these models can be time-consuming and error-prone activity [Len93]. The modelbase is stored in several tables (3 in the current implementation) because, as we pointed out in Section 5.3, the data stored in it is unnormalized. The Cassandra$^+$ system interacts with the modelbase by using *dynamic SQL* [Ing89, EN90] since it is necessary to formulate SQL queries against the modelbase dynamically "on-the-fly."

In Section 2, we claimed that the core query language can be implemented in any temporally ungrouped historical query language. This is the case because it does not matter for Cassandra$^+$ to which temporal database simulation trace files are converted and how temporal (core) queries are evaluated on these databases since these queries are passed to the temporal database as text strings without parsing.

However, in the current implementation of Cassandra$^+$ we cannot easily switch from one

database to another, as we can do it for the simulation languages. For example, it is not possible to switch from Ingres to Oracle in the current implementation of Cassandra$^+$ without rewriting (small) portions of its code handling dynamic SQL. This is because the database is used in Cassandra$^+$ for four reasons. First of all, it is used for storing simulation traces and historical information and for asking queries about them. Secondly, it is used for the *internal* implementation purposes to support modelbase. In particular, Cassandra$^+$ uses dynamic SQL for generating "on-the-fly" queries in order to retrieve data from the modelbase and for passing the core-query to the database to be evaluated. Since the implementation of dynamic SQL is systems dependent, we cannot automatically switch from one DBMS to another. Furthermore, the conversion routines from ASCII trace files to the database format are also system dependent. For these reasons, we have to provide some modifications to the Cassandra$^+$'s code when we move from one DBMS to another in the current version of Cassandra$^+$. However, these modifications are relatively small and are quite "local" to the code. Therefore, we believe that they can be easily automated in the future.

7 Conclusions

In this paper, we described a Query-Driven Simulation system Cassandra$^+$ that allows end-users to ask various questions about outcomes of simulations. We presented its query language SimQL, its architecture, and described how Cassandra$^+$ was implemented. One of the important features of Cassandra$^+$ is that it can support *any* temporal relational query language asked about simulation models written in *any* simulation language as long as trace files generated by these models conform to a certain standard.

Query-driven simulations provide more declarative, flexible, and interactive ways of asking questions about simulation outcomes than the traditional simulate-and-gather-statistics approaches. They allow end-users to ask various questions in a declarative query language in an ad-hoc manner "on the fly", just as relational query languages allow the users to ask questions about the data stored in databases.

References

[AM89] M. Abadi and Z. Manna. Temporal Logic Programming. *Symbolic Computation*, 8:277–295, 1989.

[BC84] J. Banks and J.S. Carson. In W.J. Fabrycky and J.H. Mize, editors, *Discrete-Event System Simulation*. Prentice-Hall Inc., Englewood Cliffs, New Jersey 07632, 1st edition,

1984.

[BDMR90] R. Belanger, B. Donovan, K. Morse, and D. Rockower. *MODSIM II Reference Manual.* CACI, 1990.

[Bla92] Blanning, R. and Whinston, A. and Ai-Chang, M. and Dhar, V. and Holsapple, C. and Jarke, M. and Kimbrough, S. and Lerch, J. and Prietula, M. Model management systems. In Edward A. Stohr and Benn R. Konsynski, editors, *Information Systems and Decision Processes.* IEEE Computer Society Press, 1992.

[BT93] P. Balasubramanian and A. Tuzhilin. Query Driven Simulation for Decision Support. *Working Paper (Stern School of Business, NYU)*, (IS-93-38), 1993.

[CCT93] J Clifford, A Croker, and A. Tuzhilin. On the completeness of query languages for grouped and ungrouped historical model. In A. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev, and R. Snodgrass, editors, *Temporal Databases.* Benjamin Cummings, 1993.

[Con87] Consolidated Analysis Centers, Inc. *UNIX SIMSCRIPT II.5 User's Manual*, 1987.

[Dat77] C.J. Date. *An Introduction to Database Systems.* Addison-Wesley, 2nd edition, 1977.

[EN90] R. Elmasri and S. Navate. *Fundamental of Database Systems.* The Benjamin/Cummings Publishing Company, 2nd edition, 1990.

[FDJG+92] K. Fordyce, R. Dunki-Jacobs, B. Gerard, R. Sell, and G. Sullivan. Logistics Management System (LMS): An Advanced Decision Support System for Dispatch or Short Interval Scheduling. *Production and Operations Management*, 1(1):70–86, Winter 1992.

[Geo87] A.M. Geoffrion. An Introduction to Structured Modeling. *Management Science*, 33(5):547–588, May 1987.

[IBM70] IBM. *General Purpose Simulation System/360 User's Manual*, 1970.

[Ing89] Ingres. *INGRES/OpenSQL Reference Manual for the UNIX and VMS Operating System.* Relational Technology Inc., 1989.

[Int85] IntelliCorp, Mountain View, Calif. *The SIMKIT System: Knowledge-Based Simulation Tools in KEE*, 1985.

[Kam68] H. Kamp. *On the Tense Logic and the Theory of Order.* PhD thesis, UCLA, 1968.

[Koy90] R. Koymans. Specifying real-time properties with metric temporal logic. *Journal of Real-Time Systems*, 2, 1990.

[Kro87] F. Kroger. *Temporal Logic of Programs*. Springer-Verlag, 1987. EATCS Monographs on Theoretical Computer Science.

[Len93] M.L. Lenard. A Prototype Implementation of a Model Management System for Discrete-Event Simulation Models. In *Proceedings of the 1993 Winter Simulation Conference*, pages 33–39, 1993.

[LK91] A.M. Law and W.D. Kelton. In James L. Riggs, editor, *Simulation Modeling and Analysis*. McGraw-Hill Book Company, 2nd edition, 1991.

[MP92] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag, 1992.

[MW89] J.A. Miller and O.R. Weyrich. Query Driven Simulation Using SIMODULA. In *Proceedings of the* 22^{nd} *Annual Simulation Symposium*, 1989.

[MWS90] W. Mendenhall, D.D. Wackerly, and R.L. Scheaffer. *Mathematical Statistics with Applications*. PWS-KENT, 4th edition, 1990.

[NA88] S. B. Navathe and R. Ahmed. TSQL – a language interface for history databases. In C. Rolland, F. Bodart, and M. Leonard, editors, *Temporal Aspects in Information Systems*, pages 109–122. North-Holland, 1988.

[PSS90] C.D. Pegden, R.E. Shannon, and P.P. Sadowski. *Introduction to simulation using SIMAN*. McGraw-Hill, New York, 1990.

[SAS89] SAS Institute, Raleigh, NC. *SAS User's Guide*, 1989.

[Sno87] R. Snodgrass. The temporal query language TQuel. *ACM Transactions On Database Systems*, 12(2):247–298, 1987.

[Sto86] M. Stonebarker. *The INGRES Papers: Anatomy of a Relational Database System*. Addison Wesley Publishing Company, Inc., 1986.

[TB93] A. Tuzhilin and P. Balasubramanian. Query Driven Simulation: Issues and Solutions. 1993. (in preperation).

[TC90] A. Tuzhilin and J. Clifford. A temporal relational algebra as a basis for temporal relational completeness. In *Conference on Very Large Databases*, pages 13–23, 1990.

[Tuz92] A. Tuzhilin. SimTL: A Simulation Language Based on Temporal Logic. *TRANSACTIONs of The Society for Computer Simulation*. 9(2):086–099, 1992.

[Tuz93] A. Tuzhilin. Applications of temporal databases to knowledge-based simulations. In A. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev, and R. Snodgrass, editors, *Temporal Databases*. Benjamin Cummings, 1993.

[Wir85] N. Wirth. *Programming in Modula-2*. Springer-Verlag, 3rd edition, 1985.