Speculative Query Processing

Neoklis Polyzotis
Computer Sciences Department
University of Wisconsin-Madison
alkis@cs.wisc.edu

Yannis Ioannidis
Department of Informatics and Telecommunications
University of Athens, Hellas
yannis@di.uoa.gr

Abstract

Speculation is an every day phenomenon whereby one acts in anticipation of particular conditions that are likely to hold in the future. Computer science research has seen many successfull applications of speculation: modern processors, for example, speculate on the run-time properties of a program and decide to pre-execute instructions accordingly. We draw inspiration from these techniques and introduce speculation to query processing. Our approach is based on a visual query interface that monitors the construction of a query and takes advantage of the user 'think time'. In particular, based on the features of the partial query specified at any point, the interface prepares the database by issuing asynchronous manipulations to it that are likely to make the final query (or even queries further into the future) more efficient. Furthermore, the interface applies machine learning techniques on past user actions and builds a user-behavior model that guides speculation and deals with future uncertainty. We formalize speculative query processing as an optimization problem and derive algebraic properties of the corresponding cost model that are sufficient to address the complexities of the particular optimization. We have implemented our framework on top of an existing commercial database system and have evaluated its effectiveness experimentally, with actual user traces. Our results show that speculation outperforms normal query processing, reducing query execution time by an average of 35% and achieving performance improvements of more than 90% on certain queries.

Proceedings of the 2003 CIDR Conference

1 Introduction

Webster's defines speculation as "assuming a business risk in the hope of gain". Under speculation, we predict the occurrence of a future event and then act in anticipation of that event happening. This offers opportunities of gain: if the event indeed occurs, then we end up being prepared for it, e.g., by having performed part of the necessary work ahead of time. On the other hand, this involves some risk as well: if the event does not occur, then our actions may end up being wrong, e.g., by having performed useless work. Therefore, speculation is characterized by a trade-off between risk and future payoff.

Pre-fetching techniques represent a very successful application of speculation in computer science. Modern operating systems, for example, pre-fetch file blocks if there is evidence that an application performs a sequential scan. Similarly, modern processors utilize specialized components to speculate on the memory access patterns of an application and pre-fetch memory blocks in the processor cache. Another example of speculation in modern hardware platforms is branch prediction: whenever the program counter reaches a branch instruction (if-then-else command), the processor issues the evaluation of the branch condition but does not wait for its completion. Instead, it speculates on the boolean value of the condition and pre-executes the corresponding branch of the instruction. Experience shows that, despite the risks, speculation pays off in performance.

We draw inspiration from these techniques and use the same underlying principles to introduce speculation to query processing so that query response time is decreased. The following example illustrates our intuition.

Consider an environment where users *interactively* explore the contents of a database to identify interesting patterns in it. For simplicity, the database schema contains a single relation `employee` with three fields: `name`, `age`, and `salary`. Interaction occurs through the simple QBE-like visual query interface shown in Figure 1: there is a tabular representation of the relation and the user formulates a query by placing projection

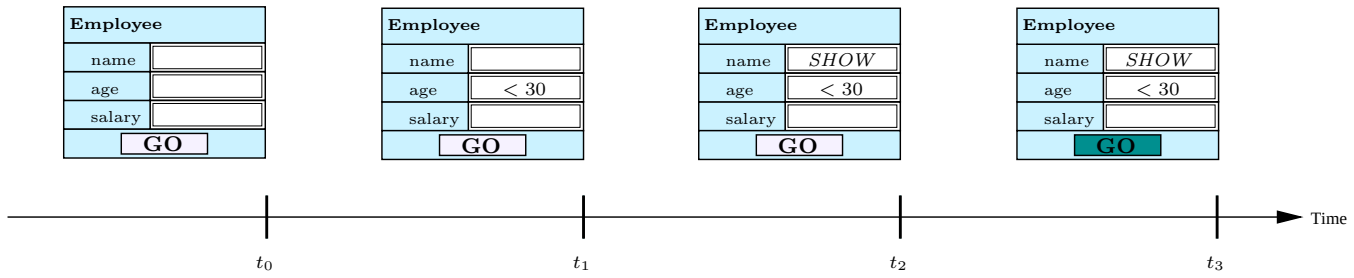


Figure 1: Query formulation

indications and selection predicates on the corresponding fields. When ready, the user clicks on the "GO" button to send the query to the database and retrieve the results. Assume that the user has the following query in mind:

```
SELECT name
FROM employee
WHERE age<30
```

Figure 1 shows how formulation of the query might progress over time, with time on the x-axis. The user places the selection predicate on `age` at time t_1, specifies a projection on `name` at time t_2, and finally, clicks the "GO" button at time t_3. We make two observations based on this trace:

• At any point before the "GO" event, a snapshot of the interface corresponds to a preview of the final query. Furthermore, the closer we are to the "GO" event, the more this preview resembles the final query.

• The user formulates the query incrementally and may spend time in various possible activities: look at earlier results and think of what the current query should be; type in the next query fragment; examine the current query for correctness or completeness. The database system is idle with respect to that user during this time and receives work (a complete query) only at the end.

These two observations form the basis of applying speculation to query processing: we use the preview of a query to speculate on its final form and take advantage of the idle time to prepare the database so that execution of the final query is more efficient. In our example, the preview of the query at time t_1 indicates that the final query is likely to contain the predicate `age<30`. Based on this, consider issuing the following materialization to prepare the database:

```
SELECT *
FROM employee
WHERE age<30
INTO TABLE young_employee
```

This is executed asynchronously, while the user is working on the rest of the visual query. Assuming that the materialization completes before the user clicks on "GO" (time t_3) and that the final query retains the predicate on `age`, we can use the new relation to rewrite the final query as follows:

```
SELECT name
FROM young_employee
```

The rewritten query produces the same answer as the original query, since the tuples of the new relation already satisfy the predicate on `age`. If there are no indices on the `employee` relation, both queries require a sequential scan: the I/O cost of the original query is `|employee|` blocks, while that of the rewritten query is $f \times$ `|employee|` blocks, where $f \leq 1$ is the selectivity factor of the predicate. As a result, we have managed to reduce the execution time of the query by a factor of $1/f$, which could be quite significant.

In this paper, we study the general concept of speculative query processing in a data exploration environment, as illustrated in the above example. More specifically, we propose the use of a speculative processing subsystem that operates within a visual query interface and takes advantage of the user's think-time in order to issue asynchronous manipulations which prepare the database for the upcoming queries. Our contributions can be summarized as follows:

• We provide a formal framework for using speculation on top of a database system, casting the problem essentially as an optimization problem. We identify and elaborate on the following parameters that determine the effectiveness of the overall approach: (a)the alternative actions that the system may take to prepare itself for an upcoming query, (b) the search strategy for enumerating these alternatives, and (c) the cost model for evaluating the expected effectiveness of each action in reducing the execution cost of the final query.

• We formulate the cost model and derive algebraic properties for cost formulas that allow speculative actions to be evaluated locally, without direct reference to all potential final queries, which are infinite in number. In addition, we embed into the cost model the ability to cache the output of speculative actions and reuse it in future queries, which in a typical exploratory environment are expected to be similar.

• We use machine learning techniques to obtain the

profile of a user with respect to how a preview of a query correlates to the final query (i.e., whether or not query parts are likely to be removed before the final query is issued) and how a query correlates to the next.

• We present the results of a realistic experimental evaluation of an implementation of our framework, involving human subjects using a simple visual interface for exploratory data analysis. These results show that certain speculative actions may greatly reduce the overall response times of queries, thus demonstrating the overall potential of speculative query processing.

To the best of our knowledge, this is the first effort that makes a strong connection between the user interface of a database system and its query processor, taking advantage of the characteristics of user interaction for reducing query response time. It has similarities with efforts like materialized view design based on a given workload or on monitoring of user queries. The key innovation is in making use of the individual user actions during query specification. Accordingly, the key benefit is in having the closest possible affinity with the user's true intentions at any one time, resulting in the best possible prepared database to answer the current query efficiently.

2 Preliminaries

Our work assumes an environment of exploratory data analysis: there is a *read-only* database and a user attempts to discover useful and interesting information in it. The process is *interactive*: the user issues a query that involves several (selection and/or join) conditions typically, visualizes the results with the appropriate presentation methods, identifies an "interesting" region of data, and proceeds with a new, usually related query. Thus, the database server receives a stream of nontrivial queries that traces the exploration path of the user. As implied by the example of Figure 1, the higher the complexity of a query, the longer the query formulation phase, and therefore the bigger the opportunity for speculative query processing to improve query response time significantly.

Our work focuses on queries that are equivalent to select-project-join queries (*conjunctive queries*). The overall formulation would remain valid for general queries as well, e.g., queries with aggregates, but some of the details would require further elaboration. In what follows, we use two representations for conjunctive queries, depending on what needs to be emphasized at each point. The first is in the form of *flat SQL* (select-from-where). The second is in the form of *query graphs*: each relation in the query is mapped to a unique vertex in the graph; each join between two relations is mapped to an edge between the corresponding relation vertices; each selection on a relation in the query is mapped to an edge between the corresponding relation vertex and a new vertex containing the operator and the constant of the selection. The vertices and edges of a query graph are the *atomic parts* of the corresponding query. Figure 2 shows an example.

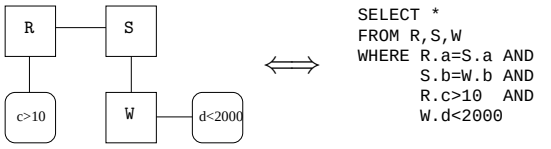


Figure 2: Example query graph

Our work assumes a visual query interface for the user interactions with the system[1]. Examples include the one implied in Figure 1, where selections are explicitly specified by users and joins are implicitly understood by the system, and one where users operate directly on query graphs. Independent of the particulars, the user builds a query incrementally, by explicitly or implicitly inserting, removing, and updating atomic query parts, and clicks on a "GO" button when ready to send it to the database. In the case of conjunctive queries, the above implies a sequence of alterations in the set of relations that participate in a query and the set of selections and joins in its qualification (vertices and edges of the corresponding query graph). At each stage, the atomic parts already specified for the query form a *partial query*. The user proceeds through a sequence of partial queries, which eventually converges to the *final query* submitted to the database after the "GO" event.

3 Speculative Query Processing

3.1 System Architecture

Figure 3 shows a schematic diagram of the system architecture that we envision. Given a database server (DBMS) and a client interface to it, we attach a speculation subsystem that monitors user actions on the interface and reacts to them appropriately. One could envision such a subsystem inside the database server as well, which would offer greater versatility and flexibility but would also require modifications to existing server code. Our general formulation is independent of where that subsystem resides. We proceed, therefore, having the diagram of Figure 3 in mind as it corresponds to what we have implemented, and we point out explicitly the places where the particular architecture imposes any restrictions. Independent of the

[1]One can envision speculation in the context of a textual query interface, although this requires further elaboration on certain aspects of our framework.

architecture choice, the speculation subsystem consists of four primary components, which are described individually below. These components are orthogonal to each other, in the sense that choices in one of them do not restrict choices in any of the others.

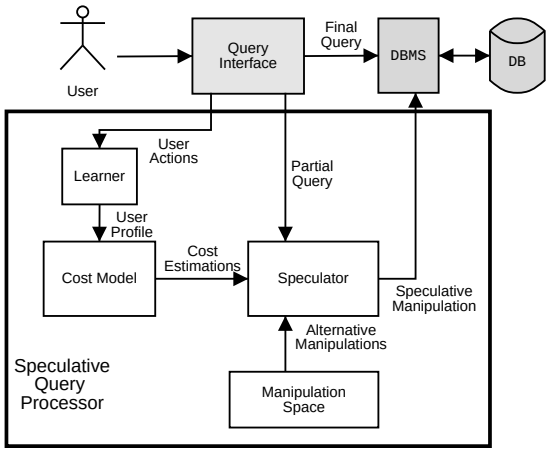


Figure 3: System Architecture

The central component of the subsystem is the *Speculator*, which is essentially a query optimizer. It accepts a partial query as input and, based on its details, it examines various alternative actions it may take to modify the state of the database; it then chooses the 'optimal' such alternative based on the cost model and sends it to the DBMS for execution.

The *Manipulation Space* is the component that determines the set of alternatives that the Speculator examines. In practice, this set is based on the current partial query that is input to the Speculator and on a small collection of available operations that can be performed on (combinations of) the atomic parts of the partial query. For the example of Section 1, materializing the selection into the temporary relation `young_employee` is one such alternative, while building an index on the `age` attribute may be another.

The *Cost Model* is the component that specifies the function optimized by the Speculator. This function represents the potential future benefits and risks of a manipulation on the execution cost of the final query when that is issued by the user. This is calculated at a time when the final query is still under development and, therefore, not known. Therefore, this function has to capture the expected benefit of a manipulation, calculated (in principle) over *all* potential final queries. One may also conceive of a version of speculation that examines the effects of a manipulation on several final queries into the future. The only difference in this case is that the optimization function has to capture the expected cost of a *sequence* of final queries (up to a certain depth).

Finally, the *Learner* observes users over time and generates profiles capturing their typical query formulation patterns. Given a partial query, the Learner uses this to generate for each potential query the probability it will indeed be the final query issued by the user, and then feeds that probability to the Cost Model.

There are three conventions on the system operation under this architecture that we have adopted and are worth mentioning. First, all speculative manipulations are executed asynchronously, i.e., while the user is working on the partial query. If the user modifies the partial query in a manner that makes the expected benefits of a manipulation under way disappear, then the manipulation is canceled. For the example of Section 1, the asynchronous materialization of the selection on `age` will be canceled if the user removes the predicate on `age`. As a special case, when the final query is issued, any manipulation currently in progress is canceled as well.

Second, a simple heuristic is used to perform garbage collection: the result of a manipulation persists as long as the current partial query indicates it will be useful for the final query. For the example of Section 1, the materialization of the selection predicate will persist after the final query (time t_3) as long as the predicate remains on the partial query. In this manner, the system takes advantage of inter-query locality.

Third, at any point the system has at most one manipulation sent to the DBMS and still outstanding (not completed), so that the overall system load is kept low.

3.2 Manipulation Space

The set of alternative manipulations generated by the Manipulation Space (Figure 3), which prepare the database for a forthcoming query, is denoted by $\mathcal{M}$. The members of $\mathcal{M}$ are defined based on a small collection of available operations applied, in principle, on any part of the database, but in practice, only on (combinations of) the atomic parts of the current partial query, i.e., on the sub-graphs of its query graph. In addition, we assume that $\mathcal{M}$ always includes a member $m_\emptyset$ that represents the null manipulation, i.e., one that does not modify the database. In our work, we consider five types of operations that define members of $\mathcal{M}$: data staging, histogram creation, index creation, query materialization, and query rewriting. We define these operations below.

Data staging warms up the buffer pool by prefetching and pinning the first few pages of a relation, saving the time to read these pages from disk if the final query accesses the relation. This technique implies the ability to pin/unpin pages in the buffer pool on de-

mand and therefore requires an interface for low-level access to the database. In effect, it is very difficult, if not impossible, to implement data staging on top of an existing DBMS (Figure 3), so we do not consider it in our implementation.

Histogram creation improves the statistics of the database by creating a histogram on an attribute of a relation. If the final query includes a predicate on the indexed field, the optimizer will produce more accurate cost estimates for the execution plans and, therefore, identify a more efficient one.

Index creation creates a new index on an attribute of a relation, providing efficient access for queries that contain predicates on the indexed field.

Query materialization generates and stores a new relation that is the result of executing a query on the database (essentially, a materialized view). Thus, the optimizer is able to use it in any final query whose graph contains the materialized query as a sub-graph.

Query rewriting is identical to query materialization, only that a materialized sub-query is always replaced by its result relation in any final query containing it. Thus, the system is forced to use the materialized relation, whereas in query materialization this was just an option. Normally, query materialization should be preferred over query rewriting: the speculative optimizer would create materialized views and the database optimizer would rewrite the query if it proved beneficial. Not all database systems, however, support materialized views and rewriting queries at the plan level; we therefore include query rewriting as a separate manipulation to cover these cases.

Each operation presents different trade-offs in how effectively it can prepare the database for the upcoming user query. Consider, for example, histogram creation and query materialization. Histogram creation has a lower execution cost and is therefore more likely to complete in time, before the final query is issued, but query materialization offers a greater potential for improvement since it creates a materialized view that can rewrite the final query. On the other hand, histogram creation is more generic and can benefit any query with a predicate on the covered attributes, while query materialization can only affect the queries that include the materialized sub-graph. In general, we can identify three characteristics that describe the trade-offs of each operation: cost of execution, impact on the final query plan, and level of specificity. As we consider the range of proposed operations, from data staging to query rewriting, we observe a trend of (a) increased cost of execution, (b) increased potential for performance improvement, and (c) increased specificity. We have verified experimentally that despite the risks, the most aggressive manipulations, i.e., query materialization and query rewriting, are the best in terms of reducing query execution time. Hence, for the remaining descriptions of the architectural components of Figure 3 we will concentrate on the specified manipulations. Some of these descriptions and formulations are more general, but this generality is not important for our discussion.

3.3 Cost Model

Let $\mathcal{Q}$ be the set of all possible (final) queries that the user may formulate. In principle, this set is infinite. Furthermore, with some straightforward conventions, any partial query may be considered as a complete query as well, so $\mathcal{Q}$ includes all the intermediate steps a user may go through during query specification. Finally, for notational convenience, let q be used to indicate both a query and its query graph. Since a query graph is a set of edges (and associated vertices), this implies that $\subseteq$, $\cup$, and $\cap$ are valid relationships and operators for queries with the natural semantics on the corresponding sets of edges and vertices.

Recall that $\mathcal{M}$ has been restricted to materializations only, so for every manipulation $m \in \mathcal{M}$, there is a query q_m that is being materialized in the database. Also, $m_\emptyset \in \mathcal{M}$ represents the materialization of nothing. For $q \in \mathcal{Q}$ and $m \in \mathcal{M}$, let $cost(q, m)$ be the (optimal) execution cost of query q on a database after manipulation m has been applied to it. The Speculator selects the manipulation $m \in \mathcal{M}$ that minimizes the expected execution cost over all possible final queries, i.e., the expected value of $cost(q, m)$ over all $q \in \mathcal{Q}$. Furthermore, this expected value weighs the cost of each query q according to its probability $f(q)$ to be the final query. Hence, if $Cost(m)$ is the function minimized, it is

$$Cost(m) = \sum_{q \in \mathcal{Q}} f(q) \times cost(q, m) \tag{1}$$

Unfortunately, evaluation of $Cost(m)$ requires enumerating $\mathcal{Q}$, an infinite set, and calculating a cost and a probability for each $q \in \mathcal{Q}$. To make the optimization problem manageable, the Speculator assumes that function $cost(q, m)$ has the following two algebraic properties:

P1 [**Containment Dependence**]: If a query does not contain a materialized query, its execution cost remains unaffected by the materialization. Formally, for query q and query-rewriting manipulation m, the following holds:

$$cost(q, m) = cost(q, m_\emptyset) \text{ if } q_m \not\subseteq q.$$

P2 [**Linearity**]: If a query consists of the union of two disjoint sub-queries, its execution cost is equal to the sum of the individual costs. Formally, for queries q, q_1, q_2 and query-rewriting manipulation

m, the following holds:

$$cost(q, m) = cost(q_1, m) + cost(q_2, m)$$
$$\text{if } q = q_1 \cup q_2 \text{ and } q_1 \cap q_2 = \emptyset.$$

Although not always valid, these properties are approximately true in many cases and have proved to be sufficient for the level of accuracy needed in the Speculator's optimization. More to the point, they lead to the following result:

Theorem 3.1 *Let $f^{\subseteq}(q)$ denote the probability that a query q will be contained in the final query. If function $cost(q, m)$ satisfies properties P1 and P2, then minimization of $Cost(m)$ defined in (1) is equivalent to minimization of $Cost^{\subseteq}(m)$ defined as follows:*

$$Cost^{\subseteq}(m) = f^{\subseteq}(q_m) \times (cost(q_m, m)\text{-}cost(q_m, m_\emptyset)) . \quad (2)$$

Based on this theorem, the Speculator needs to minimize a function that is independent of the universe $\mathcal{Q}$ of all possible queries. For a given manipulation m, the function depends on two quantities: (a) the probability that the corresponding query q_m, a subset of the current partial query, will *remain* in the final query, which is much simpler to learn than $f(q)$; (b) the difference of the cost to scan a materialized q_m from the cost to generate it from scratch, which can be computed based on standard database cost formulas. Clearly, Theorem 3.1 leads to efficiently computable cost formulas and is thus crucial in making speculation affordable.

The proof of the theorem is omitted due to lack of space. Given its critical importance, however, we present the essence of its proof through a simple example. Consider $\mathcal{Q}$ containing three queries: query $q_1 \equiv \sigma_\theta(R)$, query $q_2 \equiv R \bowtie S$, and query $q_3 \equiv \sigma_\theta(R) \bowtie S$. Further, consider $\mathcal{M}$ containing two manipulations: a materialization m_1 of (say, the current partial) query $q_1 \equiv \sigma_\theta(R)$, and manipulation $m_\emptyset$ that does not modify the database. The Speculator needs to identify the sign of the following difference:

$$\begin{aligned}
&Cost(m_1)\text{-}Cost(m_\emptyset) = [f(q_1) \times cost(q_1, m_1) + \\
&\qquad f(q_2) \times cost(q_2, m_1) + f(q_3) \times cost(q_3, m_1)] - \\
&\quad [f(q_1) \times cost(q_1, m_\emptyset) + \\
&\qquad f(q_2) \times cost(q_2, m_\emptyset) + f(q_3) \times cost(q_3, m_\emptyset)] \\
&= f(q_1) \times (cost(q_1, m_1)\text{-}cost(q_1, m_\emptyset)) + \\
&\quad f(q_2) \times (cost(q_2, m_1)\text{-}cost(q_2, m_\emptyset)) + \\
&\quad f(q_3) \times (cost(q_3, m_1)\text{-}cost(q_3, m_\emptyset))
\end{aligned}$$

If the difference is negative, then m_1 is advantageous; otherwise, it is not. Note that we can express q_3 as the union $q_1 \cup q_2$. By property P2, the original expression can then be transformed into the following:

$$\begin{aligned}
&Cost(m_1)\text{-}Cost(m_\emptyset) = \\
&\quad (f(q_1) + f(q_3)) \times (cost(q_1, m_1)\text{-}cost(q_1, m_\emptyset)) + \\
&\quad (f(q_2) + f(q_3)) \times (cost(q_2, m_1)\text{-}cost(q_2, m_\emptyset))
\end{aligned}$$

Note that q_2 does not contain q_1 as a sub-query and thus, by property P1, the second difference can be removed. Furthermore, in this simple example, $f(q_1)+f(q_3) = f^{\subseteq}(q_1)$ as q_1 and q_3 are the only queries in $\mathcal{Q}$ that contain q_1. Hence, the above expression becomes

$$\begin{aligned}
Cost(m_1)\text{-}Cost(m_\emptyset) &= f^{\subseteq}(q_1) \times (cost(q_1, m_1)\text{-}cost(q_1, m_\emptyset)) \\
&= Cost^{\subseteq}(m_1) = Cost^{\subseteq}(m_1) - Cost^{\subseteq}(m_\emptyset)
\end{aligned}$$

The last equality is from formula (2), which gives $Cost^{\subseteq}(m_\emptyset) = 0$. Clearly, the original problem has been reduced to a comparison of the values of function $Cost^{\subseteq}(m)$ for m_1 and $m_\emptyset$, as Theorem 3.1 indicates. In this example, this corresponds to a comparison of the cost of performing the selection on R ($cost(q_1, m_1)$) to the cost of accessing the materialized result of the selection ($cost(q_1, m_\emptyset)$), which is quite intuitive.

It is possible to extend the cost model so that it takes into account the effect of a manipulation on the next n user queries ($n \geq 1$), thus allowing for deeper-looking speculation. This is an important point in visual exploration environments, where consecutive user queries are similar and therefore manipulations can be re-used across multiple queries. In the full version of this paper [11], we present an extended cost formula that operates on sequences of future queries, instead of a single query, and we prove the equivalent of Theorem 3.1 for the extended case. Overall, we show that speculation on a sequence of future queries is feasible, as it requires only local references to the sub-queries corresponding to the materializations under consideration.

3.4 Learner

The Learner observes the user's on-screen actions and builds a profile that characterizes the behavior of the user during query formulation. This profile is ultimately used to provide estimates for the probability terms that appear in the cost formulas. The profile is continuously updated with information on the most recent actions of the user and is based on several learner components. Each learner captures one specific aspect of the user's querying behavior and approximates a specific probability term in the cost model. The complete details on the machine learning methods used and the training of the learners can be found in the full version of this paper [11].

3.5 Speculator

The main task of the Speculator is to enumerate the manipulations of $\mathcal{M}$ generated by the Manipulation Space, so that it can choose the one with the least value of $Cost()$ based on the details supplied by the Cost Model. As mentioned earlier, we only consider query materialization and query rewriting manipulations, since we have verified experimentally that they

are the most effective in reducing query execution time. The enumeration of all possible materializations, however, can be prohibitively expensive and can even become intractable in certain cases. As a result, we have made the following choices regarding the algorithm for generating possible manipulations:

- We consider materializations on sub-graphs of the current partial query only. We do not examine materialization of queries that are not compatible to the current partial query, since it is not likely to pay off: the partial query is highly likely to be contained in the final query and speculation on the remaining structure of the final query is rather fruitless.
- We consider materializations of individual selection edges or materializations of individual join edges enhanced with all selection edges attached to the join edge. We do not generate materializations on arbitrary sub-queries of the current partial query since examining all the sub-queries of a large partial query would be expensive and, furthermore, not all sub-queries correspond to useful materializations.

Overall, the Speculator enumerates materializations on selections and on two-way joins with all relevant selections attached. In addition, the enumeration algorithm takes into account previous materializations that have completed and that are relevant to the current partial query. Consider, for example, the partial query $\sigma_\theta(R) \bowtie S$ and assume that $R \bowtie S$ has already been materialized in a relation T. The enumeration algorithm will produce two materializations for $\sigma_\theta(R)$: $T_1 \leftarrow \sigma_\theta(T)$, which uses the already completed materialization T, and $T_2 \leftarrow \sigma_\theta(R) \bowtie S$, which re-executes the join. The two materializations have the same result but require different database operations, and the cost model will eventually select the one with the lowest execution cost. The complete details for the Speculator component, along with the pseudo-code for the enumeration strategy, can be found in the full version of this paper [11].

4 Experimental Evaluation

We have implemented a prototype speculative query processor based on the architecture of Figure 3. The database server is Oracle 8i while the client is a visual query interface that we have developed. We have conducted several experiments with this prototype. Their details are given in the following subsections and their results in the next two sections.

4.1 Experimental Methodology

In order to create a realistic simulation of our target environment in our experiments, we used real traces of human subjects exploring a dataset. We collected the traces with a simple data exploration experiment: we presented a particular dataset to fifteen users and posed five questions to them. Each question was phrased in English and was formulated at an abstract level, e.g., "Find three suppliers that are expensive, and the items they supply are either not popular or can be located by other suppliers at a lower cost". For each question, the user was free to explore the dataset and pose an arbitrary number of ad hoc queries to find a suitable response. Furthermore, we had generated a skewed dataset with certain trends and patterns, so that the users would discover meaningful answers.

The users formulated queries on the SQUID visual interface (`http://www.cs.wisc.edu/~alkis/squid`). SQUID employs a paradigm similar to QBE: the user places projection and selection annotations on the visual schema, and the visual query is then translated to a SQL query and executed against a relational DBMS. We used a modified version of the interface that recorded the timing and actions of each user in a separate trace file, which was then used to replay the user session on demand. For each experimental setup (see below), each trace was replayed twice, once for normal processing and once for speculative processing. For each replay, we measured the elapsed execution time for any query set $\mathcal{Q}_\mathcal{I}$ of interest and then evaluated the performance of speculative processing as a percentage of improvement over normal processing as follows:

$$improvement = 1 - \frac{\sum_{q \in \mathcal{Q}_\mathcal{I}} time_{spec}(q)}{\sum_{q \in \mathcal{Q}_\mathcal{I}} time_{normal}(q)}$$

A positive *improvement* represents a reduction in query execution time, while a negative value represents an increase in query execution time (compared to normal processing).

4.2 Experimental Setup

Our setup consisted of a dual Pentium-II machine with 1GB of physical memory, running Redhat Linux 6.2. Both the interface client and the database server run on the same machine. For the experiments, we replayed each trace with a cold buffer pool and with no other user entering the system.

The schema used for the dataset was a subset of the schema of the TPC-H [13] benchmark. It involved six tables (orders, customer, lineitem, partsupp, supplier, part) mutually connected through various foreign keys. It was populated with data of varying size for different experiments, and of high skew in fields that were likely to appear in selections in user queries. Finally, it was supported by indices and histograms on all skewed fields and foreign key fields so that the database was fully prepared for the experiment queries.

As mentioned in Section 3.2, we verified experimentally that query materialization and query rewriting

outperform histogram and index creation in terms of reducing query execution time. Furthermore, we focused entirely on query rewriting as query materialization would have complicated significantly the implementation of our system due to the unorthodox way the particular DBMS that we used treats non-join views.

5 User Querying Behavior

In this section, we discuss the querying behavior of users observed during the experiment. The fifteen users that participated covered a wide range of database expertise: they were one database professor, nine database graduate students and five other graduate students. None of the users was aware of what we were trying to measure or had any particular familiarity with the schema or data. In what follows, we present the main characteristics of an "average" user profile, based on a number of useful statistics. A more extended analysis can be found in the full version of this paper [11].

Query Structure: On the average, each trace generated 42 SQL queries and each query contained 1-2 selection predicates and referenced 4 relations in the FROM clause; user queries, therefore, had enough complexity to make speculation worth while. Once inserted into the partial query, a selection predicate remained unmodified for three consecutive queries on the average and a join predicate for ten; hence, reusing speculative materializations across queries increased their amortized benefit.

Query Formulation: Query formulation starts with the first modification of the visual query and ends when the user clicks on the "GO" event. During this interval, the user is idle with respect to the database and the speculative framework issues the asynchronous manipulations that will prepare the database for the upcoming query. The duration of this interval, which is indicative of the user's think-time, determines the feasibility of applying speculation on query processing. The following table shows the minimum, average, maximum, and three percentiles (25%, 50%, 75%) for the duration (in seconds) of query formulation in the collected traces:

	min	avg	max	25%	50%	75%
Duration	1	28	680	4	11	29

Overall, the results show that the users' think-time is large enough for asynchronous manipulations to be issued and complete in time.

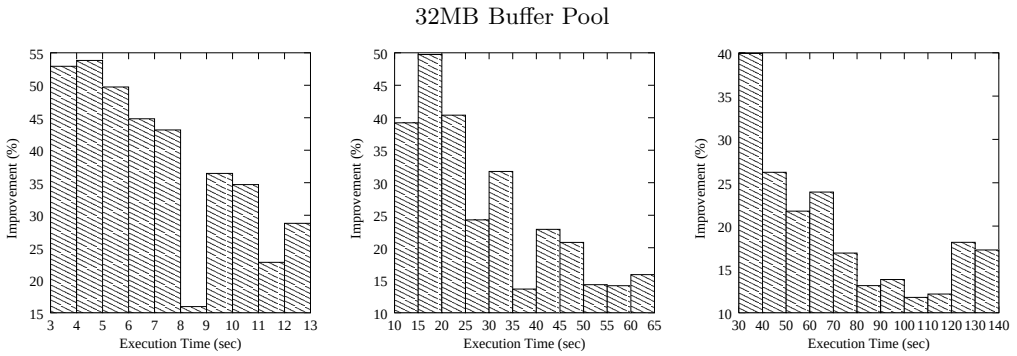
6 Results

We have experimented with three dataset sizes: 100MB, 500MB and 1GB. Throughout our experiments, we have consistently observed that the distribution of query execution time is skewed towards "short" queries: in general, users issue a few general "long" queries and then explore more thoroughly a smaller subset of the data. Based on the above observation, we have focused our attention on the initial time ranges that include the great majority of queries. We present our results in the form of a barchart: we group queries in buckets according to their execution time under normal processing and we calculate the performance metric (Section 4.1) for the queries in each bucket. Furthermore, we ensure that each bucket contains at least 5 queries so that the computed metric is statistically robust and unaffected from the effects of outliers. For the three dataset sizes, we have therefore identified the following intervals: 3-13 second for the 100MB dataset, 15-65 seconds for the 500MB dataset and 30-140 seconds for the 1GB dataset. These intervals contain the majority of queries and are used for the entire presentation. The remaining, long queries are very few to base any statistically valid results on them (fewer than 5 queries in each buckeet). Even so, their behavior is similar qualitatively to the ones discussed.

6.1 Sensitivities to Environment

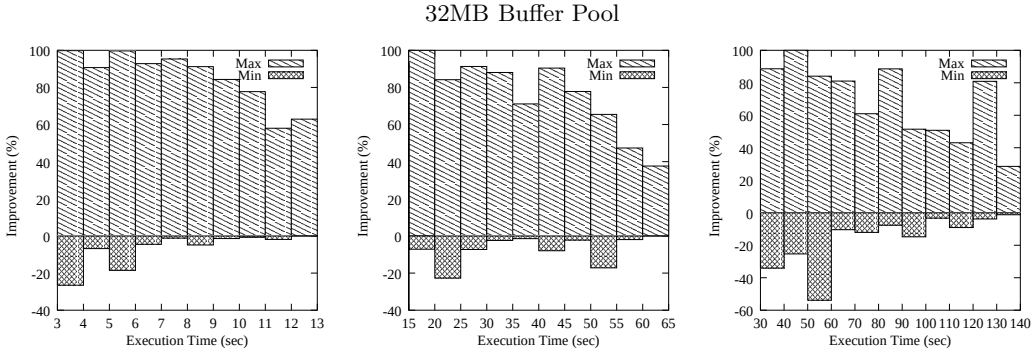
Figure 4 shows the relative performance of speculation for the three different sizes of the database mentioned above. We observe that speculation consistently improves query execution time significantly for all datasets: for the 100MB, 500MB, and 1GB datasets, it reduces query execution time by 42%, 28%, and 20% on the average, respectively (that is, normal processing is 74%, 39%, and 26% more expensive, respectively). The trend is clearly for decreasing improvement as the database size grows. Part of the reason is that, in our experiment, a materialization needs on the average 6 seconds on the 100MB dataset, 9 seconds on the 500MB dataset and 10 seconds on the 1GB dataset. As a result, the percent of manipulations that do not complete in time grows with the size of the dataset: 17% for 100MB, 25% for 500MB and 30% for the 1GB dataset. At the same time, however, completed manipulations have a greater effect on query execution time and thus speculation still performs well due to the increased savings in execution time.

The extreme effects of speculation are also interesting. Figure 5 shows the maximum performance improvement and the maximum penalty for each bucket of queries. Overall, speculation may offer substantial improvements and may reduce query execution time by almost 100% for certain queries. In the 1GB dataset, for example, speculation can yield sub-second response time for a query that needs 40 seconds under normal processing. Maximum penalties, on the other hand,



(a) 100MB Dataset (b) 500MB Dataset (c) 1GB Dataset

Figure 4: Average relative performance of speculation under different dataset sizes



(a) 100MB Dataset (b) 500MB Dataset (c) 1GB Dataset

Figure 5: Maximum performance improvement/declination under different dataset sizes

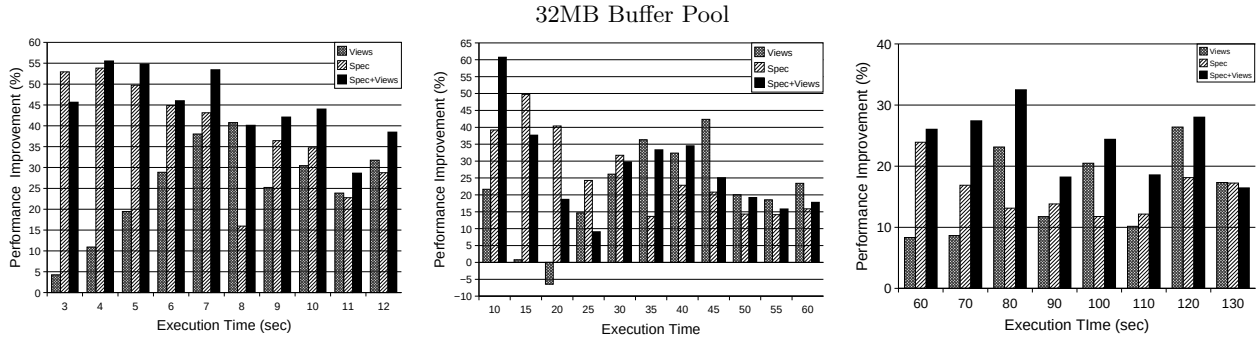
are much smaller than the improvement benefits and become significant in few cases only; even then, they mostly involve "short" queries and therefore the absolute time penalty is not dominant. The source of such negative performance is almost invariably the creation and use of a materialized relation without an index in place of an existing relation that is sorted or has an efficient index on it. Although the Cost Model takes all these issues into account, occasionally its estimates lead to wrong decisions.

We have also performed several experiments with memory-resident databases, which we do not present here due to lack of space. Overall, our results show that materializations can reduce execution time significantly even if they do not reduce I/O cost, and thus speculation continues to outperform normal query processing when the database is memory resident.

6.2 Speculation vs. Materialized Views

In this section, we evaluate the performance of speculation relative to materialized views. View materialization is a popular precomputation technique and, as such, it represents an alternative to speculation. On the other hand, the two approaches are mutually orthogonal, in the sense that speculation can be applied on both databases with materialized views and databases without such views. In order to explore all possibilities, we have experimented with three separate runs of the traces: the first uses speculative query processing on top of no materialized views (same runs as before); the second uses normal query processing on top of materialized views; and the third uses speculative query processing on top of materialized views. In our experiments, we have materialized the join of each possible subset of the database relations, keeping all their attributes. This configuration represents an extreme case in favor of materialized views, as the DBMS can answer any query by scanning the appropriate view and possibly applying selections only, without accessing the base relations or performing any joins. Normally, storage constraints would limit the number of created views and several queries would not be able to benefit from any view.

Figure 6 shows the performance of the three approaches as an improvement over normal processing without materialized views (Section 4.1). Overall, speculation offers more improvement for shorter queries, while materialized views tend to perform better as queries become more costly. These queries in-



(a) 100MB Dataset (b) 500MB Dataset (c) 1GB Dataset

Figure 6: Average relative performance of speculation vs. materialized views vs. their combination

volve large table joins and speculative processing can only complete a limited number of the relevant manipulations; on the other hand, materialized views allow the system to answer the query with a single scan instead of executing the join. The combination of the two techniques, however, seems to be the winner in all but a few cases, sometimes decisively so, demonstrating the "universal" value of speculation. In the few cases where the combination loses to pure materialized views, the reason is identical to that mentioned in the previous section for the case where speculation loses to normal processing. Also, in the few cases where the combination loses to pure speculation, the reason is invariably misjudgement on the part of the DBMS optimizer and use of a materialized view that is not as efficient as using the original relations. The same reason holds for the few cases where pure materialized views lose to normal processing.

In the current formulation of our framework, the speculative optimizer is ignorant of the existence of pre-materialized views and issues the same manipulations as in the previous experiments. A promising direction would be to make the speculative optimizer aware of the existing pre-materialized views. With a modified cost model and manipulation space, it could create new materializations on top of the existing views, allowing the database optimizer to rewrite the final query even more efficiently. We plan to explore this option in our future work.

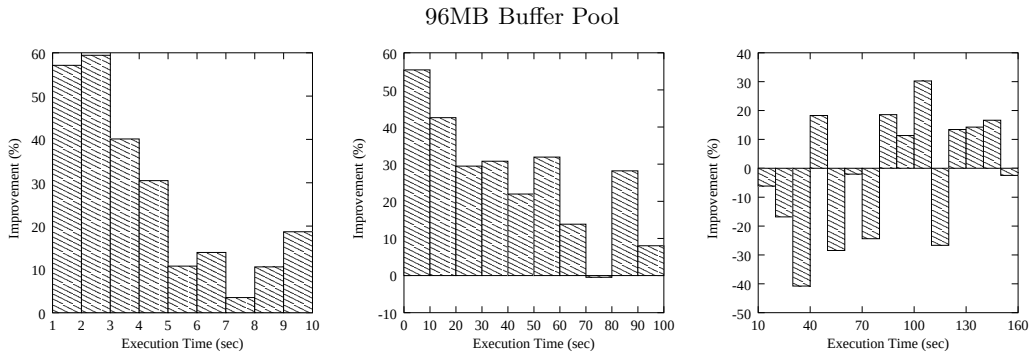
6.3 Multi-user Experiments

In the previous experiments, we have evaluated performance in a single-user environment: we have replayed each trace individually when no other users are accessing the database. We now present the results of a limited set of experiments that evaluates speculation in a multi-user setting. The cost model remains as is, not evaluating the impact of speculative manipulations on speculative or even regular query processing by other concurrent users. In any other case, the cost model would have become extremely complicated, requiring information on all users of the system, and could very quickly make speculative query processing not worth while. Despite the lack of global knowledge by the cost model, however, our results show that speculation continues to outperform normal processing.

We evaluate the performance of speculation in a multi-user environment by replaying simultaneously three different user traces. In order to reduce the interference of speculative manipulations on the queries of other users, we have employed a modified enumeration strategy (Section 3.5) that generates materializations of selection predicates only. In this manner, we execute simpler manipulations and thus reduce the additional load on the system. We assign 96MB to the buffer pool, which represents a scale-up proportional to the number of users.

Figure 7 shows the performance of speculation in a multi-user setting. We have modified the time ranges on the x-axis compared to the single user case, in order to account for the different execution time of queries. The results show that, again, speculation improves query performance for most queries, especially in the 100MB and 500MB datasets, although as expected, less so than in the single-user case. We also observe some nontrivial penalties for several queries in the 1GB dataset: in these cases, the server is already under a very high load during normal query processing and the additional overhead of asynchronous manipulations results in significant penalties. Still, the performance improvements are a rather promising indication of the effectiveness of speculation in a multi-user environment.

Speculation gives promising results in a multi-user setting, even if the cost model does not account for other concurrent users. In general, we expect that speculation will become increasingly less effective as the number of users grows. A natural solution would be to suspend speculative processing when the server is busy, and resume it when the load falls below a



(a) 100MB Dataset (b) 500MB Dataset (c) 1GB Dataset

Figure 7: Average relative performance for three simultaneous users

threshold. We plan to explore this option plus others in our future work.

7 DBMS Support for Speculation

Up to this point, we have focused on an "external" implementation of our speculative framework, as a middle-ware module between the interface and the database system. Throughout our experiments, however, we have observed several opportunities where a tighter integration, between the speculator and the database, could yield increased performance. In what follows, we describe the type of functionality that a DBMS could provide, in order to better support speculative query processing.

Current database systems do not provide any feedback on the remaining time for a running materialization. As a result, the Speculator follows a conservative approach and cancels all incomplete manipulations when the final query becomes known, instead of waiting for their completion and then using their results to rewrite the final query plan. If the DBMS could provide an estimate on the remaining time to completion for pending manipulations, the Speculator could evaluate the benefit of delaying the execution of the final query until certain materializations complete.

An alternative approach would be for the DBMS to use the partial results of incomplete materializations in the final query plan. Consider, for example, an incomplete materialization of a join $R \bowtie S$ into a new relation T and assume that $R \bowtie S$ is contained in the final query q. It might be more efficient if the execution plan uses T to scan the initial part of $R \bowtie S$ and then evaluates the remaining part from the base relations. Overall, this solution suggests a different model of speculative processing, where the speculator essentially builds partial execution plans and the database system "stitches" them together when the final query is issued.

Finally, the DBMS could provide functionality that would make speculation more efficient in a multi-user environment. In order to help the Speculator reduce the overhead of asynchronous manipulations, the database could provide information on the current system load and available resources, so that materializations are issued when the system is not busy. Another approach would be to provide support for query scheduling based on priorities, so that manipulations are issued with lower priority compared to normal user queries. In this manner, materializations run only when there are available resources and are suspended when user queries need to be executed.

8 Related Work

Query caching and self-tuning systems represent applications of speculation in database systems. In query caching [3, 9, 12], the database uses the results of previous queries to answer efficiently similar subsequent queries; in essense, this technique speculates that the next query of the user will resemble the previous ones. Self-tuning systems [2, 4, 14] operate in a similar fashion: they monitor the incoming queries and use the collected information as the basis for speculating on the properties of the workload. Based on the speculated properties, they tune the database accordingly by creating the appropriate indices and materialized views [1, 7], pre-fetching data pages [10], or modifying operating parameters [5].

The key difference between our approach and the two mentioned above is that they operate at the final query level and do not consider the individual actions of the user during query formulation. In query caching and self-tuning systems, speculation occurs once the final query becomes known; in effect, speculation does not take in account the actions of the user in between and good performance depends on the final queries providing a sufficiently representative workload. Our technique, on the other hand, operates at a much finer level and targets explicitly the stage of query formu-

lation. Speculation occurs whenever the partial query is modified and is based both on the previous queries and the current state of the query under development. As a result, speculation should be able to make better informed decisions on the structure of the final query, based on the most recent and most highly-correlated information there can be: parts of the query itself already specified (even if tentatively). Another characteristic feature of our approach is that it incorporates user-interface aspects into speculation: based on the particular, exploratory style of human-DBMS interaction, speculative query processing takes advantage of a user's "think"/idle time to precompute and prepare the database. We are not aware of any other work that has linked database user interfaces with query processing in such beneficial fashion. Finally, the Learner trains on the query formulation actions of the user and builds a profile that feeds into the necessary probabilities of possible final queries. Again, we are not familiar with any other work on profiling users at such a fine level of behavioral detail.

Online Query Processing [8] represents another example of employing user interface interaction to affect query execution. In online processing, the user visualizes the partial results of a running query and desires different levels of detail over certain regions of the data. The system then attempts to guide query execution so that it refines faster the partial results over the interesting regions. Online processing, therefore, considers user-interaction during *query processing* (the final query is already known), whereas our technique operates during *query formulation* (the final query is not yet known). The two techniques are in fact orthogonal and can be used in combination. This also raises the opportunity of using speculative processing in order to facilitate online processing, e.g., by materializing ripple joins [6]. We plan to investigate this direction in our future work.

9 Conclusions

In this paper, we have introduced the concept of speculative query processing. We have presented a framework that operates within a visual query interface and issues asynchronous actions to prepare a database for future user queries. We have formulated speculation as an optimization problem and we have presented a cost model that deals with uncertainty and abstracts the lack of knowledge for the final query over a set of possible queries along with a probability for each query. Although the general formulation of the problem is intractable, we have derived cost expressions that are efficient to compute and have proposed the use of machine learning techniques to estimate the needed probabilities. We have presented an evaluation of our prototype implementation, using simulation traces that we collected from real exploratory sessions with human subjects. Our results have shown that speculation outperforms normal query processing under various database configurations, improving query performance by an average of 35% and in some cases even by 90%. We have also presented a limited number of multi-user experiments and showed that speculation continues to outperform normal processing in many cases even if it does not deal explicitly with multiple concurrent users.

References

[1] Agrawal, S., Chaudhuri, S., and Narasayya, V. R. Automated selection of materialized views and indexes in sql databases. In *Proceedings of 26th International Conference on Very Large Data Bases* (2000).

[2] Chaudhuri, S., and Weikum, G. Rethinking database system architecture: Towards a self-tuning risc-style database system. In *Proceedings of 26th International Conference on Very Large Data Bases* (2000).

[3] Deshpande, P., Ramasamy, K., Shukla, A., and Naughton, J. F. Caching multidimensional queries using chunks. In *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data*.

[4] Drew, P., King, R., and Hudson, S. E. The performance and utility of the cactis implementation algorithms. In *Proceedings of the 16th International Conference on Very Large Data Bases* (1990).

[5] Faloutsos, C., Ng, R. T., and Sellis, T. K. Predictive load control for flexible buffer allocation. In *Proceedings of the 17th International Conference on Very Large Data Bases* (1991).

[6] Haas, P. J., and Hellerstein, J. M. Ripple joins for online aggregation. In *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data*.

[7] Hammer, M., and Chan, A. Index selection in a self-adaptive data base management system. In *Proceedings of the 1976 ACM SIGMOD International Conference on Management of Data*.

[8] Hellerstein, J. M., Haas, P. J., and Wang, H. Online aggregation. In *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*.

[9] Kamel, N., and King, R. Intelligent database caching through the use of page-answers and page-traces. *TODS 17*, 4 (1992).

[10] Palmer, M., and Zdonik, S. B. Fido: A cache that learns to fetch. In *Proceedings of the 17th International Conference on Very Large Data Bases* (1991).

[11] Polyzotis, N., and Ioannidis, Y. Speculative query processing. Tech. rep., University of Wisconsin-Madison, 2002.

[12] Sellis, T. K. Intelligent caching and indexing techniques for relational database systems. *IS 13*, 2 (1988).

[13] Transaction Processing Performance Council. *TPC-H Benchmark Specification*.

[14] Yu, C. T., and Chen, C. H. Adaptive information system design: One query at a time. In *Proceedings of the 1985 ACM SIGMOD International Conference on Management of Data*.