The Researcher's Guide to the Data Deluge: Querying a Scientific Database in Just a Few Seconds

Martin L. Kersten, Stratos Idreos, Stefan Manegold, Erietta Liarou

CWI, Amsterdam, The Netherlands
{mk,idreos,manegold,erietta}@cwi.nl

ABSTRACT

There is a clear need for interactive exploration of extremely large databases, especially in the area of scientific data management where ingestion of multiple Terabytes on a daily basis is foreseen. Unfortunately, current data management technology is not well-suited for such overwhelming demands.

In light of these challenges, we should rethink some of the strict requirements database systems adopted in the past. We envision that next generation database systems should *interpret queries by their intent*, rather than as a contract carved in stone for complete and correct answers. The result set should aid the user in understanding the database's content and provide guidance to continue the data exploration journey. A scientist can stepwise explore deeper and deeper into the database, and stop when the result content and quality reaches his satisfaction point. At the same time, response times should be close to instant such that they allow a scientist to *interact* with the system and explore the data in a contextualized way.

Several research directions are carved out to realize this vision. They range from engineering a novel database kernel where speed rather than completeness is the first class citizen, up to refusing to process a costly query in the first place, but providing advice on how to reformulate it instead, or even providing alternatives the system believes might be relevant for the exploration patterns observed.

1. INTRODUCTION

The Challenge of Extremely Large Data Sets. Scientific databases face an ingestion of Terabytes of data on a daily basis. This data becomes useful information only after in-depth analysis using the methods deployed in science, i.e., validation of models and induction of rules from observations. Even nowadays, scientific groups already struggle with the basics, i.e., to store their data or even to move it around, let alone to analyze it in an interactive mode.

Database technology has evolved with a different paradigm in mind. It mostly targeted financial applications, where *correctness and completeness* are key, and where there is a vast amount of a priori knowledge to prepare the system for fast response. In scientific databases though, none of this is true anymore. Other than the vast amount of data to be processed, the users do not always know exactly what they are looking for and they not always care for a complete answer; it is mainly used for data *exploration* in search for interesting patterns. In this paper, we revisit some of the hitherto strict restrictions set in database system design and envision alternative research paths to handle the challenge of analyzing extremely large scientific data sets.

Interactive Data Exploration. For example, consider the success of web search engines. For a large part they rely on guiding the user from his ill-phrased queries through successive refinement to the web-pages of interest. Limited a priori knowledge is required. The sample answers returned provide guidance to drill down chasing individual links or to adjust the query terms. Recent approaches in facet-based information retrieval are an attempt to further speed-up the search by providing summary information.

Compare this ease of use with how database systems are queried. It requires the user to learn a query language, know its database schema, and control the amount of output being generated. Once a query is submitted, the system will blindly obey the user's orders and do its utmost best to provide a complete and correct answer as quickly as possible. It returns anything from an empty set to the complete database. Exploration speed and effectiveness can be improved by the user using database statistics, sampling, synopsis, and prior knowledge. Pre-canned queries and materialized views aid new users in finding their way in the query space.

The situation in scientific databases however, is even more complicated, because they often contain complex observation data, e.g., sky images or seismograms, and little a priori knowledge exists. The prime challenge is to find models that capture the essence of this data at both a macro- and a micro-scale. Paraphrased, the answer is in the database, but the Nobel-price winning query is still unknown.

For example, consider the >1 TB PhotoObj table in the Sloan Digital Sky Survey with around 500 columns, most of which are floating point numbers, and >600 M rows. Scientific proof for a hypothesis comes from finding data correlations, e.g., spatially clustered objects with similar properties, and checking data obtained from simulation of astrophysics phenomena in this database. Such data exploration is the prime domain of data mining, where statistical methods and learning algorithms are the predominant tools being used. Unfortunately, finding new rules and patterns works well only in high quality datasets, i.e., those that have undergone a thorough quality assurance check. Outliers and noise are often suppressed, while in science those can be the more interesting parts to gain insight. For example, in recent years, detailed analysis of the noise in seismograms revealed that some plates are actually

Proceedings of the VLDB Endowment, Vol. 4, No. 12

moving up and down, like a bed spring. Such results are found by generating a hypothetical database state through simulation and search for it in the collected (dirty) observations.

Vision. Next generation query processing engines should provide a much richer repertoire and easier to use querying techniques to cope with the deluge of observational data in a resource limited setting. Good is good enough as an answer, provided the journey can be continued as long as the user remains interested.

In this short note, we propose a few alternative system design paths which if realized can significantly improve the user experience in exploring scientific databases. To set the stage, consider the simplistic SkyServer query Q: *select * from PhotoObj*. The SkyServer query logs illustrate that such generic queries are actually being used. Asking for a sizable part of the database may also result from accidental erroneous modification of pre-canned queries using too broad parameter ranges or hitting a highly dense area in the database. Running query Q against the 4 TB SkyServer database held in Microsoft SQL Server or MonetDB nowadays would return a completely incomprehensible answer of millions of rows with floating point numbers, only cut off by the page-size limit set in the reporting tools and the 10 minute time out for query processing. Its complete answer may even take days to produce. The situation becomes worse if queries start to return only empty sets. Clearly, this would not be the user's intent in most situations. Instead, a more informative answer should be given. But how can we achieve this?

The underlying cause for a long running query with unwanted large (or empty) result set is the user's "ignorance". The user simply can not a priori know how to phrase the query in such a way that it is both fast to process with limited resources and still produces informative answers. It is this lack of providence that should be addressed, much like information retrieval techniques in search aim for user satisfaction, despite the lack of completeness and correctness.

In the remainder of this paper we charter a visionary landscape of potential ground-breaking research. In particular, we consider five concrete tracks into this open space:

- One-minute database kernels for real-time performance.
- Multi-scale query processing for gradual exploration.
- Result-set post processing for conveying meaningful data.
- Query morphing to adjust for proximity results.
- Query alternatives to cope with lack of providence.

There does not exist a free lunch in science either. Therefore, to make this vision become a reality, the user should at least identify the budget he is willing to spend on query processing. For the remainder we assume his waiting time for a response to be the critical factor, but result quality or energy consumption can also be considered budgetary items to aim for. Like search engines the challenge is *what can the system provide within T seconds?* A rethinking of the database system architecture along this line and a possible overhaul of major portions may be required.

The remainder of the paper provides a short outlook on the system we have in mind. We focus on the total picture of a new system architecture for scientific data exploration, built on the strong foundation of database technology.

2. ONE-MINUTE DB KERNELS

One of the prime impediments to fast data exploration is the query execution focus on correct and complete result sets, i.e., the semantics of SQL presupposes that the user knows exactly what he needs from the system. The design and implementation of the query optimizer, execution engine, and storage engine are focused towards this goal, helped by proper database indexing, partitioning, data distribution, and parallel execution. That is, correctness and completeness are first class citizens in modern DB kernels. This means that when the system needs to perform a few hard unavoidable steps (e.g., random access, I/O, etc.), it is designed to perform them such that it can produce the complete and correct results. But is it needed in the case of scientific databases with Petabytes of data?

If the user accidentally produces a large result set, then a sample might be more informative and more feasible. Unfortunately, such a sample depends on the data distribution, the correlations, and data clustering in the database and the query result set. And taking a sample can still cause significant performance degradation that surface only at run time. An experienced user would resort to querying a pre-computed database summary first. For scientific databases though, even creating such summaries on the daily stream of Terabytes becomes a challenge on its own.

The first proposal is to address the problem at its root; we envision *one-minute database kernels* that have rapid reactions on user's requests. Such a kernel differs from conventional kernels by trying to identify and avoid performance degradation points on-the-fly and to answer part of the query within strict time bounds, but also without changing the query focus. Its execution plan should be organized such that a (non-empty) answer can be produced within T seconds.

Although such a plan has a lot in common with a plan produced by a conventional cost-based optimizer, it may differ in execution order, it may not let all relational operators run to completion, or it may even need new kinds of operators. In other words, a one-minute kernel sacrifices correctness and completeness for performance. The goal is to provide a quick and fully interactive gateway to the data until the user has formulated a clear view of what he is really searching for, i.e., it is meant as the first part of the exploration process.

Let us indicate a few research directions. For example, large scans over base tables are bound by the I/O capacity of the underlying system hardware. For a given T this puts an upper-bound on the total amount of data that can be read from/written to the persistent store. Moreover, database queries often contain blocking operations that lead to a pipeline stall or spilling large intermediates back to the disks. They should be avoided, or replaced by less accurate versions. To illustrate, when building a hash table during a join, we can choose to not insert certain elements such as to keep the size of the hash table within the memory or within the cache. Similarly, while probing the hash table we may choose not to follow long linked lists to avoid cache misses.

In addition, very often during a plan we need to sort large sets of rowIDs to guarantee sequential data access. Those can be replaced by a cheaper clustering method or we can refrain from data access outside the cache. Likewise, operations dealing with building auxiliary structures over the complete columns/tables, can be broken up into their piecewise construction. Building just enough within T to make progress in finding an answer. If T is really short, e.g., a few seconds, the plan may actually be driven from what is already cached in the memory buffers. In a modern database server, it is just too expensive to free up several GB of dirty memory buffers before a new query can start. Instead, its memory content should be used in the most effective way. In the remaining time the memory (buffer) content can be selectively replaced by cheap, yet promising, blocks from the disks. With a time budget for processing, the execution engine might either freeze individual operators when the budget has been depleted, or it might replace expensive algorithms with approximate or cheaper alternatives.

These ideas extends from high level design choices in database operators and algorithms all the way to lower level (hardware conscious) implementation details. For example, during any database algorithm if we reach the case where we need to extend, say, an array in a column-store with a realloc, an algorithm in the one minute kernel may choose to skip this step if it will cause a complete copy of the original array.

This sketch is just the tip of the ice berg, i.e., numerous examples and variations can be conceived. The key challenge is to design a system architecture where budget distribution can be dynamically steered in such a way that the query still produces an informative result set. At first sight, traditional Volcano-style processing schemes may seem to come a long way to provide this functionality. Until one actually would go in and consider the administrative burden to control global scheduling, blocking operators and incremental indexing. Aside from a tedious large-scale (re-)engineering effort to build a kernel on this assumption, major research questions arise. For example: How is the budget spread over the individual operators? What actions are database operators allowed to take to stay within the budget? How to harvest the system state produced by previous queries? How to replace the relational operators and index constructors with incremental versions?

3. MULTI-SCALE QUERIES

Our second proposal towards fast exploration of large datasets is to aim for a *staging scheme*, where stepwise a larger portion of the database becomes the query target. As the user becomes more and more confident on the direction of his exploration he is willing to analyze more data and spend more of his budget.

Most large scale scientific databases are a priori partitioned, and the database fragments directly correlate with an external file format held in a repository. For example, the major seismic events are kept in a repository of $> 3 * 10^6$ files. Similarly, data gathered with the Large Hadron Collider at CERN needs huge file repositories of $> 10^8$ files, and also the database design for the Large Synoptic Survey Telescope expects millions of large database partitions.

In these situations, we can rethink the cost-based optimizers, deployed in contemporary systems, to exploit this inherent partitioning. Instead of strictly finding the optimal plan assuming the complete database, we can break the query into two pieces $Q = Q_1 \cup Q_2$, such that one piece can be evaluated within the bounds of the user budget using a limited number of the total partitions/files. This technique can be applied recursively, while staging can be applied both horizontally and vertically. Even simple solutions can provide very useful functionality. For example, in a column store, where tuple reconstruction is an expensive component, simply limiting the number of columns in the target list is a good first step. If needed, the user can always ask for more columns and the system has to keep track of the proper intermediates to quickly resume processing. Keeping track of what portions have been touched makes it possible to continue afterwards, and even give a more accurate prediction on the total query execution time.

The key challenge is to find the cost-metrics and database statistics that allow the system to break queries into multiple steps. Optimization techniques may focus on preparatory subqueries and passing intermediates between the stages.

4. RESULT-SET POST PROCESSING

For decades, database designers have largely ignored the potentials of post-processing a result set. At best, the user can steer sorting the results or limit the number of rows to be returned. But, is this all we can do? And should we obey a sorting over hundreds of TB? To provide even more informative answers may call for stepping even further away from the target expression in a SQL query. The guiding example query Q with millions of rows could perhaps be compressed into a more meaningful way. Straightforward compression in the SkyServer context may not work, for most of the data consists of hard-to-compress floating point sequences.

But, there are many statistical techniques that can be processed in linear time over the result set. For that, the result set is not shown to the user as is, but passed through a straightforward statistics post-processor to derive informative results. Min, max, and mean values for all attributes are the first that come to mind. For larger target lists, the data distribution may be further elicited by selective sampling. Histograms could also be constructed in such a way that the number of bins shown fits the screen, while the bounds are derived based on the value deviation within each bin.

A minimalistic post-processing action is to look at the result set and sample it in such a way that the contours of the result set value distribution become visible. It is conceivable that two tuples suffice to express the min-max values for all attributes. This information aids the user in cutting out portions for drill down.

Post-processing the result set should ideally retain their structure, otherwise they might confuse front-end applications expecting a certain number of columns as part of their JDBC interaction protocol. However, the standard graphical client interfaces could be informed through a simple protocol on the result set structure. Tool tips can be provided on the actions taken for further clarification.

Of course, both query processing and result set processing should be handled within T. This further complicates the query optimizer, because it has to also estimate the post-processing time or piggyback information gathering in the critical path of query execution.

5. QUERY MORPHING

The directions seen so far target performance improvements regarding the queries posed. In a science exploratory setting though even choosing the proper queries becomes a challenge. Even with a time-aware kernel and data exploration with multi-stage querying, badly chosen queries will still significantly hinder data exploration. Perhaps they are formulated wrongly, or they get stuck in a part of the database where no satisfying results exist.

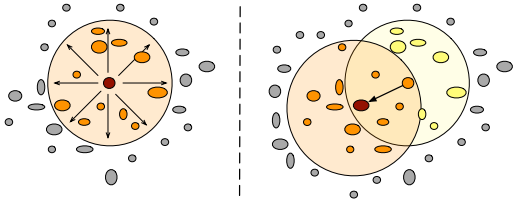

Figure 1: An example of query morphing.

For this recurring situation, we introduce the notion of *query morphing* as an integral part of query evaluation. It works as follows, the user gives a starting query Q and most of the effort T is spent on finding the "best" answer for Q. But a small portion is set aside for the following exploratory step. The query is syntactically adjusted to create variations Q_i, i.e., with a small edit distance from Q. The process of query morphing is visualized on the left part of the above figure. The user's original request returns the result set depicted by the small red circle. However, the database kernel grabs the chance to explore a wider query/data spectrum in parallel, providing additional results for queries that belong in the *close area*, surrounding the original request. The arrows that start from the red circle indicate this edit area in our example. This way the user also receives the orange elliptic query results that correspond to variations of his original request. In the right part of above figure,

we see that the user may as a next step decide to shift his interest towards another query result, inspired by the result variations. A new query area now surrounds the user's request, including both past and new variations of the query.

Several kinds of adjustments can be considered to create the query variations, e.g., addition/dropping of predicate terms, varying constants, widening constants into ranges, joining with auxiliary tables through foreign key relationships, etc. The kind of adjustments can be statistically driven from queries ran in the past, or exploitation of database statistics gathered so far, or even cached past (intermediate) results. Since we have already spent part of our time on processing Q, the intermediates produced along the way can also help to achieve cheap evaluation of Q_i.

The approach sketched aligns to proximity-based query processing, but it is generalized to be driven by the query edit distance in combination with statistics and re-use of intermediates. Query morphing can be realized with major adjustments to the query optimizer, because it is the single place where normalized edit distances can be easily applied. It can also use the plan generated for Q to derive the morphed ones. The ultimate goal would be that morphing the query pulls it in a direction where information is available at low cost. In the ideal case, it becomes even possible to spend all time T on morphed queries.

6. QUERIES AS ANSWERS

No matter how well we deploy the techniques sketched so far, they still do not address the user's lack of knowledge about the data. With huge data sets arriving on a daily basis, scientists do not always have a clear view on the data characteristics or even what they are looking for.

It seems prudent to propose that the database system *itself* can provide starting points and guidance for data exploration; instead of returning results for a random or badly formulated query, the system can return *query suggestions for more effective exploration*.

Compared to query morphing discussed before, here the system returns interesting or popular queries that are believed to produce meaningful result sets. In the case of query morphing, the system takes an opportunistic stance and automatically returns auxiliary results around the area of the user's interest that happen to be collected at reasonable cost.

Assume a query with an excessive time to execute, and even tens of continuation steps, e.g., the estimated response time is $> 2^7\,T$ or the result set $> 42\,\%$ of the database. In this situation, the system might simply "refuse" the query and instead call for more precision from the user. Every refused query should lead to an advisory list of query alternatives with an indication on how large their result set would be when run within T. For example, the advisory list for query Q introduced in Section 1 could be:

```
-- Q1: Using the time budget. (36291322 tuples)
    SELECT ra, dec, band1, intensity1, type
    FROM PhotoObj;

-- Q2: Using data statistics. (879300 tuples)
    SELECT * FROM PhotoObj
    WHERE ra BETWEEN 53 AND 54
    AND dec BETWEEN 80 AND 82;

-- Q3: Using query statistics. (899 tuples)
    SELECT * FROM PhotoObj
    WHERE ra BETWEEN 53 AND 54
    AND dec BETWEEN 80 AND 82
    AND distance(ra,dec,radius) < 10;
```

In essence, the original query Q says "tell me everything you know about table PhotoObj". This is naturally a very expensive operation over Terabytes of data. On the other hand, the advisory query $Q1$ illustrates that only a small portion of the persistent database can be retrieved within T seconds. $Q2$ shows a query carving a smaller region of the sky (from the SkyServer logs it is known that the positional lookups are the most frequently executed). $Q3$ refines it further to a manageable subset.

There are two crucial research challenges here. First, we should be able to identify "bad" or "wrongly formulated" queries. Second, we need to identify "interesting" queries to return as an answer.

For the first part, a bad query can be identified by the optimizer provided good statistical information is available. In case of scientific databases though, even creating such statistics becomes a huge challenge due to the massive data sets arriving on a daily basis. Instead, we need mechanisms to on-the-fly detect bad, i.e., very expensive queries, and cancel or pause their execution. The user can then provide feedback on how to proceed, e.g., resume execution of the current query or continue with one of the queries in the advisory list. An expensive query can be detected more easily at run time due to the more accurate information, i.e., intermediate cardinalities, selectivities, etc. Another option is that query execution simply pauses when the user budget expires or the user can request an advisory query list immediately without even posing a query.

For the second part, there are several ways to aid in creating the query advisory list. For example, for scientific applications such as the SkyServer, there exist traces of millions of past queries. Based on such query logs, we can derive a set of queries, which are used frequently, or for which the users have provided a positive feedback, or whose result set is expected to cover part of the original query. The latter is, of course, hindered by the limitations to derive an expression subsumption relationship. However, for exploratory scenarios guidance rather than precision is more valuable.

To further improve the response, we probably need an enriched vocabulary. For example, the SkyServer database comes with several tens of user defined functions that encode astronomical concepts. The user experience would improve if a system could exploit this set in its suggested alternatives. It could "pollute" a query using a weighted sampling over the domain-specific set of user defined functions and exploit a semantic web -like interlinking to aid the user in understanding the implications of using them.

7. SUMMARY

In this short vision paper, we have shown that there are at least five different directions where the user experience in querying a scientific database can be significantly improved. Directions geared at rebuilding a database kernel better suited for incremental processing, stepwise multi-scale query processing to drill down into the haystack, straightforward statistical analysis and data mining over result sets for summaries, patching queries for proximity, and the extreme approach to answer a query with just a set of alternative queries. Scientific progress in these areas does not necessarily require starting from scratch, but a different viewpoint on how to use/adapt the algorithms and techniques we know already.

8. ACKNOWLEDGMENTS

Many researchers have inspired us in describing the challenges identified here. Their work on query processing, approximate query processing, cache conscious algorithms and so many other areas of research in the database field formed the stepping stones to form the current vision. Attribution to only a few would not do justice to the database research community at large.