The SIGMOD 2010 Programming Contest A Distributed Query Engine

Clément Genzmer [1] Volker Hudlet [2] Hyunjung Park [3]
Daniel Schall [2] Pierre Senellart [4]

[1] Facebook, USA [2] TU Kaiserslautern, Germany
[3] Stanford University, USA [4] Télécom ParisTech, France

ABSTRACT

We report on the second annual ACM SIGMOD programming contest, which consisted in building an efficient distributed query engine on top of an in-memory index. This article is co-authored by the organizers of the competition (Clément Genzmer, Pierre Senellart) and the students who built the two leading implementations (Volker Hudlet, Hyunjung Park, Daniel Schall).

1. CONTEXT

For the second year in a row, a programming contest was organized in parallel with the ACM SIGMOD 2010 conference. Undergraduate and graduate student teams from over the world were invited to compete to develop an efficient distributed query engine over relational data. Students had several months to work on their implementation, which was judged for their overall performance on a variety of workloads. The teams responsible for the five best systems were invited to present their work during the SIGMOD 2010 conference, and the winning team (one-man team *cardinality* formed of Hyunjung Park, Stanford University), was awarded a prize of $5,000.

In addition to encouraging students to be active in the database research community, the aim is to build over the years, throughout a series of contests, an open source in-memory distributed database management system. Thus, the candidates of this year's contest relied on the in-memory index implementation produced as the outcome of last year's competition.

We first describe in more detail the task the contestants were involved in, as well as the workload their implementation was evaluated on. We then report on the outcome of the competition, before describing the key ideas of the systems ranked first and second.

2. TASK DESCRIPTION

As previously mentioned, the task was to program a simple distributed relational query engine. Contestants had to provide a binary library conforming to a specific interface, along with the corresponding source code. Each submission was evaluated on a dedicated cluster of eight machines, over a series of eight secret query loads. The input provided to the implementation for each workload was the description of which nodes of the cluster stored (parts of) which tables, possibly horizontally partitioned, as well as a set of queries, expressed in a simple subset of SQL. The goal was then to provide the correct output to these queries, as fast as possible. The final score of each submission was computed as a monotonous function of the total time used for running all workloads. Workloads where the submission crashed, did not return the correct output, or ran over the time limit of five to ten minutes (depending on the workload), were assigned penalties.

All queries were simple select-project-join queries, of the form:

```
SELECT alias.attribute, ...
FROM table AS alias, ...
WHERE condition1 AND ... AND conditionN
```

where a condition might be any of:

- `alias.attribute = constant`
- `alias.attribute > constant`
- `alias1.attribute1 = alias2.attribute2`

A parser for this subset of SQL was provided.

Attribute values were either character strings or integers, and tables were stored in text files on disk. All tables had at least one column indexed in memory, the implementation of the index being provided based on last year's contest. Before the actual starting of each workload, the contestants were given a predefined number of seconds to perform some preprocessing steps over the data. At this point, their implementation received a set of queries which was representative of the workload.

Among the eight benchmarks, five were of a reasonably small scale and were designed to test the workability of the binary provided by the contestants. The last three were designed to test the performance (up to 150,000

queries, and up to 1,000,000 tuples stored on a given node). Contestants were given a one-line description of each workload, though the actual structure of the input data was not disclosed. The benchmarking tool would initiate 50 parallel connections on a master node and then would start issuing the queries.

The full description of the task is available at `http://dbweb.enst.fr/events/sigmod10contest/`.

3. THE COMPETITION

The initial description of the task was made available in December 2009 along with all necessary interfacing code, though some addition and fixes came over the following months as bugs or imprecisions were pointed out by contestants. Starting from February 2010, contestants had access to the evaluation cluster and could check the score of their submission and their ranking. Students had then up to April to work on their implementation. Then, a shortlist of five finalist teams, whose submission had the best performance, was selected and these teams could use the remaining time before the beginning of the conference, in June, to continue improving their implementation.

Setting up the evaluation cluster, eight dedicated PCs running a 64bit version of Linux on a single-core CPU, was not completely straightforward. First, it was critical to ensure the security of the machines and the non-disclosure of the contents of the benchmarks, whereas contestants were allowed to run arbitrary code on the cluster. The solution chosen was to run each submission as a new unprivileged Linux user without any write access to any part of the disk (except for a temporary directory that was emptied after each evaluation), and to set up a strict firewall that prevented any information leaking outside the cluster. Second, for scores to be meaningful, evaluation times had to be reproducible from one run to another. This led us to use dedicated servers rather than virtualization, to clear all system caches across runs, and to ensure, as much as possible, that no concurrent processes were active on the cluster nodes. Third, it turned out that contestants encountered problems (crashes, timeouts) while running their submissions on the evaluation workloads that they were not able to reproduce on their test benchmarks. To help them with debugging, we provided them with stack traces and other similar crash information, from the execution log of the evaluation. This, however, could not be automated, because of potential leaks of detailed information about the content of the benchmarks, and resulted in a time-consuming task for the organizers.

A total of 29 teams took part in the competition, from 23 different institutions in 13 countries. An amazing collective effort was put in this contest, with some teams literally dedicating months of working time to the competition. As a result, leading implementations are impressively sophisticated and efficient.

The five finalist teams and their score, computed at the end of the competition, are listed in Table 1. Note that only the top two teams managed to pass all benchmarks in the allocated time. Both of these implementations are interesting: the winning one, *cardinality*'s, is generally faster, but is actually slower than the second-ranked, namely *dbis*'s, on the last workload, which is also the most difficult one.

More details about the results of the competition can be found at the following URL: `http://dbweb.enst.fr/events/sigmod10contest/results/`.

4. KEY IDEAS BEHIND LEADING IMPLEMENTATIONS

We present in this section the general ideas behind the two systems we implemented. When not stated otherwise, the description applies to both.

Given the task, query planner and executor are two main components to build. Undoubtedly, both of them can make an enormous impact on the overall performance. Here we discuss key decisions on designing and implementing these components in more detail. We refer to a textbook on database management systems such as [2] for precisions on some of the techniques used.

Query Planning

In order to determine the most efficient physical query plan, we implemented cost-based query optimization. Our objective is to minimize total resource consumption rather than response time because of the many concurrent queries running at the same time. The cost model focuses on network transfer as well as sequential and random disk reads. Incorporating CPU cost would be a natural next step, but we believe that its impact on the quality of the selected plan will be marginal given the performance of the main-memory index and the limited class of supported SQL queries. Also, calibrating parameters for complex models would have been more difficult due to restricted access to the evaluation cluster.

Finalist teams employed various plan search strategies. Team *cardinality* enumerates all possible physical query plans and estimates their costs. Because plans are built bottom-up, a dynamic programming technique is applied in order to avoid building and evaluating the same subplan multiple times. During plan enumeration, uninteresting subplans are pruned according to heuristics that favor index-based operators. These heuristics not only reduce the size of search space but also complements the rather simple cost model. Note that time and space consumption of this exhaustive search is bounded because the contest specifies that at most five tables are joined. On the other hand, team *dbis* first performs an exhaustive search for all possible join orders and then refines the

Table 1: Final score, corresponding time, and proportion of the eight workloads processed

	Team	Institution	Country	Score	Time	Workloads
1.	*cardinality*	Stanford University	USA	88	3min 18s	8/8
2.	*dbis*	TU Kaiserslautern	Germany	98	5min 45s	8/8
3.	*spbu*	Saint-Petersburg University	Russia	108	>12min 17s	7/8
4.	*insa*	INSA Lyon	France	119	>22min 20s	7/8
5.	*bugboys*	KAUST	Saudi Arabia	142	>30min 19s	5/8
	Naïve implementation			207	>45min 31s	2/8

plan with optimal join order using heuristics. Once the optimal join order is chosen, the query planner starts with a basic plan using table scans and block nested-loop joins. Subsequently, alternative plans are derived using as many index-based operators as possible. In this step, a block nested-loop join can be replaced by a merge join when two primary key columns are joined. Likewise, if one or both join columns are indexed, a hash join can be used by treating the given index as a hash-based access structure. A similar procedure applies for access operators; a table scan is substituted by an index scan followed by an optional table seek based on the offset retrieved from the index.

Exploiting partitioning information during query planning turns out to be a crucial observation for passing all benchmarks. Clearly, certain queries can be answered by accessing only relevant partitions because all tables are range-partitioned based on the primary key column. For example, primary key joins do not cover all possible partition pairs, but are processed only between partition pairs whose ranges overlap. Similarly, conditions on the primary key column can eliminate out-of-range partitions. Because accessing multiple partitions usually involves network transfer, skipping one partition can decrease the running time significantly. If partitioning information indicates that a condition is unsatisfiable, we generate a query plan with a no-op operator that produces an empty result set.

Statistics of each partition are gathered during the preprocessing step. We obtain partition sizes and estimate the cardinality and average column sizes by sampling a few first pages in each partition. Also, we count the number of distinct values for indexed columns. Other than these basic statistics, we do not maintain detailed information about data distribution and rely on fixed selectivity factors depending on the type of predicates.

Query Execution: Single Node

Tables stored on disk are accessed through memory-mapped I/O. This strategy has two main advantages. First, we can deploy a high performance buffer pool with little implementation effort because the page cache in the Linux kernel provides this function for us. Even though a naïve approach using an input file stream library also utilizes the page cache, we cannot read the page cache directly. Consequently, excessive memory copy between kernel and user spaces cannot be avoided. Second, we can access columns spanning multiple pages efficiently. Unlike most traditional databases, underlying data files are stored in the comma-separated values (CSV) format. Thus, addressing a column would be complicated unless there is a contiguous address space for an entire file. Note that despite these advantages, this strategy would not have been practical on 32-bit architectures where available address space is considerably smaller.

Query execution uses a simple pull-based, Volcano-style, iterator model [1] where each operator implements Open(), GetNext(), and Close() interfaces. A tuple passed by GetNext() is always represented as a vector of pointer-length pairs. Because we avoid memory copy as much as possible, these pointers usually reference either memory-mapped files or communication buffers.

Query Execution: Multiple Nodes

Send and receive operators fulfill the communication among the master and slave nodes. A receive operator generates a request message by serializing the subplan rooted at its child and ships the message to its corresponding send operator. Upon receiving a request message, the send operator constructs the subplan, executes it, and sends its result back as a response message. Each send operator runs on its own worker thread for concurrent execution. Some finalist teams maintained a thread pool with a fixed number of worker threads. However, a naïve approach without a thread pool also performs well because only hundreds of threads are created and destroyed each second in this setting.

The send and receive operators exchange messages over TCP/IP connections, so efficient TCP communication is vital to the performance. The first challenge to minimize TCP overhead is to amortize TCP connection setup and tear-down cost across multiple request-response pairs. Finalist teams addressed this challenge in slightly different ways, but the main idea is to maintain established TCP connections for reuse. Some teams keep a single TCP connection between each pair of nodes,

while the others initiate a new TCP connection whenever there is no reusable connection. In the contest setting, the latter works better because receive operators do not have to wait for the single TCP connection to be available. Another challenge is to reduce per-message TCP overhead in order to increase application throughput. To achieve this goal, we pack each message into fewer TCP segments by setting the `TCP_CORK` option on all TCP connections. The Linux kernel does not transmit partial TCP segments as long as the `TCP_CORK` option is set and a 200-millisecond timer does not expire. As a result, more network bandwidth is utilized by request and response messages rather than TCP/IP headers. These two techniques yield a substantial performance improvement; benchmark 6 runs more than four times faster.

Message compression is another natural way to save network bandwidth. For request messages, we adopted a variable-length integer code that serializes smaller integers into fewer bytes. This simple encoding is quite effective for request messages because serialized query plans contain many small integers such as node ids and column ids. An optimized variable-length encoding implementation decreases the running times of benchmarks 6 and 7 by more than 20%. For response messages, we experimented with a couple of lossless compression algorithms. Even though we observed good compression ratios and some performance improvements using our test dataset and queries, we failed to decrease any benchmark running time. This result is not surprising because the cost of message compression can be more expensive than the benefit depending on workload and hardware configuration. Clearly, we need a more sophisticated algorithm that determines which messages to compress in order for this technique to work well across various workloads.

5. CLOSING REMARKS

The SIGMOD 2009 and 2010 programming contests were a chance for many students (at both the undergraduate and graduate levels) to discover and design parts of the architecture of a distributed database management system. The contest was used in several universities as part of the curriculum or as an optional alternative to other assignments. This programming contest will run again next year, organized by Stavros Harizopoulos and Mehul Shah from HP Labs. It is our hope that this competition will help foster the next generation of database researchers and practitioners.

6. ACKNOWLEDGMENTS

We are very grateful to the sponsors of the programming contest: NSF, Microsoft (platinum sponsors); Amazon, INRIA Saclay (gold sponsors); Exalead, Yahoo! (silver sponsors). We would also like to acknowledge our advisory board: Serge Abiteboul, Magdalena Balazinska, Samuel Madden, and Michael Stonebraker.

7. REFERENCES

[1] Goetz Graefe. Volcano - an extensible and parallel query evaluation system. *IEEE Trans. Knowl. Data Eng.*, 6(1):120–135, 1994.

[2] Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems*. McGraw-Hill, New York, USA, third edition, 2002.