Crescando

Georgios Giannikis[*1] Philipp Unterbrunner[*2] Jeremy Meyer[†3]

Gustavo Alonso[*4] Dietmar Fauser[†5] Donald Kossmann[*6]

[*]Systems Group, Department of Computer Science, ETH Zurich, Switzerland
[1,2,4,6]giannikg,philippu,alonso,kossmann@inf.ethz.ch

[†]Amadeus IT Group SA, France
[3,5]jeremy.meyer,dfauser@amadeus.com

ABSTRACT

This demonstration presents Crescando, an implementation of a distributed relational table that guarantees predictable response time on unpredictable workloads. In Crescando, data is stored in main memory and accessed via full-table scans. By using scans instead of index lookups, Crescando overcomes the read-write contention in index structures and eliminates the scalability issues that exist in traditional index-based systems. Crescando is specifically designed to process a large number of queries in parallel, allowing high query rates. The goal of this demonstration is to show the ability of Crescando to a) quickly answer arbitrary user-generated queries, and b) execute a large number of queries and updates in parallel, while providing strict response time and data freshness guarantees.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems—*Parallel databases, Query processing, Distributed databases*

General Terms

Experimentation, Performance, Measurement

Keywords

Main Memory, Parallel Database

1. INTRODUCTION

A key challenge for enterprise-scale on-line data management systems is multi-dimensional growth. As technology and companies evolve, these systems need to grow not only in terms of throughput, but also in terms of query diversity and complexity, due to increasing numbers of users supported, and number of services provided. As a result, the data management infrastructures of industries such as financial, insurance, or the travel industry grow simultaneously in multiple dimensions, making it increasingly difficult to meet service level agreements (particularly response times) and data freshness constraints for all services and corresponding queries.

Existing solutions to this problem are secondary indexes and redundant materialized views. Graefe[2] gives a comprehensive survey of all the tricks researchers have devised in the last three decades to trade read performance and data freshness for write performance. However, none of these techniques can change the fact that indexes are a shared resource, and they all come at a price in terms of system complexity, storage, and computational overhead. To make matters worse, materialized views come with additional administrative cost to setup and maintain. In any real system there will always be a point at which materializing yet another view or creating yet another index is no longer economical.

What is sought for is a simple, highly scalable solution to growth in *all* dimensions: query throughput, query diversity, and update rate. The goal is an affordable system able to answer *any* query on-line, with bound latency, on live data under a heavy update load.

To this extent, we have developed a novel system, called Crescando[4], that is able to overcome the multi-dimensional growth problem by making a radical simplification. Instead of using many materialized views and indexes, turn the system into a main-memory distributed database which processes queries exclusively via continuous memory scans. This approach sidesteps the view selection and index maintenance problems. Simply speaking, it takes read-write contention and query diversity out of the equation, reducing multi-dimensional growth to a throughput problem, which can be solved by adding hardware alone.

In this demonstration, we showcase Crescando's performance on a table of 60 attributes containing a total of 360 million records, based on a real-world dataset from our partner Amadeus S.A. The demonstration allows the conference participants to experiment with Crescando's ability to answer a) user-defined complex queries and b) a vast number of concurrent queries and updates while maintaining strict latency and data freshness guarantees.

SIGMOD'10, June 6–11, 2010, Indianapolis, Indiana, USA.

2. CRESCANDO

In this section we outline the main features of Crescando and its query model. Additionally, we present the distributed architecture and a bottom-up overview of the 3 tiers that it consists of. Further details about Crescando and our implementation were recently published in [4].

2.1 Overview

Crescando is a relational table implementation in which data is stored in main memory and accessed exclusively via scans. Crescando horizontally partitions the dataset and assigns each partition to a scan thread which is bound to a dedicated CPU core. Scans are performed in parallel: one scan thread per CPU core. There is a strict shared-nothing policy between scan threads, which eliminates hot spots in the system. Finally, Crescando takes advantage of the latest non-uniform memory architectures (NUMA) to maximize performance of individual scan threads and achieve linear scale-up in the number of available CPU cores.

2.2 Features

In Crescando, an operation is either a *query* (simple SQL-style `SELECT` statement with optional scalar aggregation) or an *update*. The term update is used for any unnested, SQL-style `INSERT`, `UPDATE`, or `DELETE` statement.

Crescando's scan threads run a novel algorithm, called Clock Scan, to both query and update the data. Clock Scan allows multiple operations to be evaluated by a single full table scan, which improves cache locality. Additionally, Crescando performs query-data joins using Index Union Join, an algorithm inspired by publish/subscribe systems and also presented in [4]. Index Union Join indexes operations with predicates on the same attributes in one-dimensional index structures. As a result, the access latency of Clock Scan and respectively Crescando is logarithmic in the number of operations that are being executed at a given moment.

Index Union Join is able to index and execute thousands of operations in a single memory scan. Crescando benefits from executing very large numbers of queries in parallel, as this invokes the Law of Large Numbers. The more operations are executed at the same time, the more they are representative of the workload as a whole, giving stable performance and balancing the load across the system.

The current implementation of Crescando supports operations whose selection predicate can be expressed as a conjunction of predicates of the form *attrib op const*, where *attrib* is an attribute of the relation, *op* is a comparison operator, and *const* is a constant value. An example is:

$$\sigma_{AirportFrom='JFK',Birthday<'1.1.1940'}(R)$$

Queries containing disjunctions can always be expressed as a union of such queries, so this is not a limitation.

2.3 Distributed System Architecture

Since main memory is limited, a single machine might be unable to store the entire table. So for scalability and also availability, Crescando utilizes a distributed architecture based on horizontal data partitioning and replication. The basic principles are presented in [1], but we present them here for completeness.

Crescando partitions the table between *replication groups*, which consist of *storage nodes*, whose data is accessed in read-one write-all (ROWA) fashion. The replication groups form the *storage layer*. Inside each storage node, the dataset is further partitioned, and each partition is assigned to a dedicated CPU core that runs the Clock Scan algorithm.

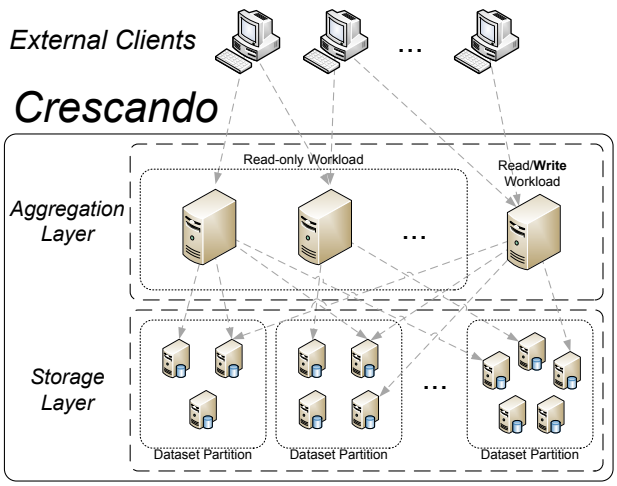

Figure 1: Crescando Distributed System Overview

One or more layers of *aggregator nodes* are responsible for routing operations to replication groups, and merging ("aggregating") the results. Figure 1 visualizes the distributed architecture. It is similar to that of the NDB storage engine used by MySQL Cluster [3], to name just one example.

3. DEMONSTRATION OVERVIEW

In this demonstration, we showcase the performance of Crescando through an interactive demo. The demo highlights Crescando's performance on a large, real-world dataset of airline ticket reservations, extracted from the production database of Amadeus SA, a leading provider of IT solutions to the tourism and travel industry. The demo consists of two parts: a feature demo and a performance demo.

The demonstration performs operations over a dataset of more than 360,000,000 airline ticket reservations. These represent all flight bookings made over a period of two years. Each record is roughly 300 bytes, thus the whole dataset adds up to around 100 GB of net data.

3.1 System Setup

The demonstrations consists of 8 storage nodes, each built from 2 quad-core Intel Xeon L5520 "Nehalem", clocked at 2.26 GHz with hyperthreading enabled, and 24 GB of RAM. This adds up to a total of 120 CPU cores for scanning data (1 CPU core per machine is reserved for network I/O). We limit the amount of data per CPU core to 1 GB, allowing the storage of up to 120 GB of data. Additionally, there is one aggregation node, running on a dedicated machine with said specifications.

During the demonstration, a small pool of laptops, one of them connected to a projector, will allow the conference participants to interact with our instance of Crescando by issuing arbitrary queries (cf. Figure 2). This forms the "feature" part of our demo. At the same time, the conference participants will be able to specify a synthetic background load on the system, and observe how this affects query and

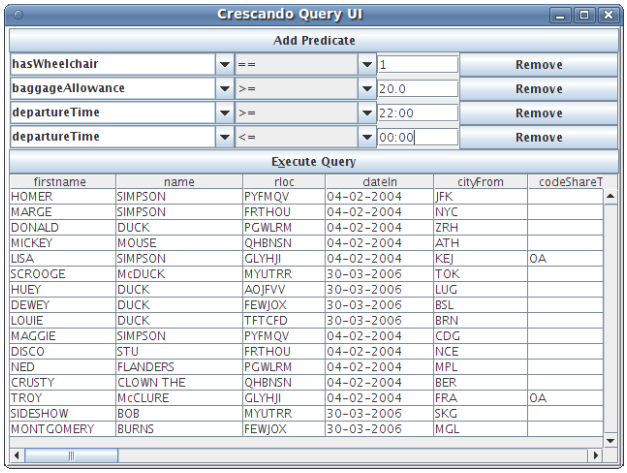

Figure 2: Screenshot of Crescando's Feature Demonstration

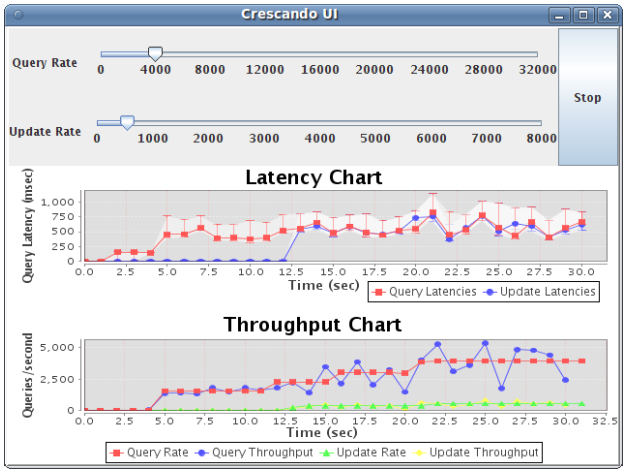

Figure 3: Crescando's Performance Demonstration - Query Throughput and Latency Window

update latency (cf. Figure 3). This forms the "performance" part of our demo.

The laptops will be connected to a server cluster of 9 nodes, running at ETH Zurich. This setup naturally requires an Internet connection. In the event of bad internet connectivity, we will run an appropriately scaled-down instance of Crescando on the laptops themselves.

3.2 Feature Demo

The feature demo allows a participant to compose any query he or she desires, and execute it against the full dataset. The interface allows for creation of queries that filter on any of the 60 attributes of the table schema, using any of the 6 comparison operators supported. The number of predicates per query is not limited.

After the user has submitted a query, he or she can observe how Crescando populates the result table after a near-constant amount of time, regardless of the nature and number of predicates. The following are a few example queries under which a traditional index oriented system does not exhibit predictable performance, while Crescando is able to answer with strict latency guarantees:

- "What is the number of passengers that require a vegetarian meal in flights that departs before 7am from Heathrow airport."
- "What are the names of the passengers that checked in at 15:00 for a flight that departed at 15:30."
- "What are the names of the passengers that require a wheelchair, have a baggage allowance of more than 20Kg and fly after 22:00."

Figure 2 is a screenshot of the proposed query interface along with an exemplary result set.

3.3 Performance Demo

The second part of the demonstration allows the conference participants to observe how Crescando reacts to varying load and varying query diversity. As regards load, the interface provides two sliders: one for query rate, and one for update rate. The interface continuously measures and plots

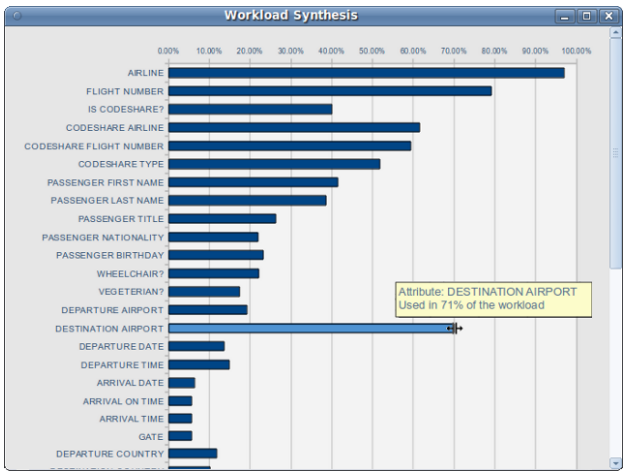

Figure 4: Crescando's Performance Demonstration - Workload Synthesis Window

the median, as well as the lower and upper bound, of query and update latency. A screenshot of this interface is shown in Figure 3.

As regards query diversity, the demonstration provides a separate window for synthesizing an arbitrary workload. This window, shown in Figure 4, allows the user to define how likely any attribute is to appear in a query.

The performance demo makes it easy to investigate:

- the relation between the number of concurrently executed queries and the query latency,
- the friendliness of Crescando to concurrent queries and updates, and
- the friendliness of Crescando to diverse workloads.

4. ACKNOWLEDGMENTS

This work has been funded in part by the Amadeus IT Group SA, as part of the Enterprise Computing Center of ETH Zurich (`www.ecc.ethz.ch`). The work of G. Giannikis

and P. Unterbrunner is funded in part by a Swiss National Science Foundation grant as part of its Pro-Doc program. Part of this work was done during G. Giannikis' internship at Amadeus IT Group SA.

5. REFERENCES

[1] G. Giannikis. Daedalus, a distributed crescando system. Master's thesis, Swiss Federal Institute of Technology Zurich, April 2009.

[2] G. Graefe. B-tree indexes for high update rates. *SIGMOD Rec.*, 35(1):39–44, 2006.

[3] M. Ronström and L. Thalmann. Mysql cluster architecture overview: High availability features of mysql cluster. MySQL Technical Whitepaper, 2004.

[4] P. Unterbrunner, G. Giannikis, G. Alonso, D. Fauser, and D. Kossmann. Predictable performance for unpredictable workloads. In *Proc. VLDB '09*, 2009.