

Predictable Performance for Unpredictable Workloads

P. Unterbrunner* G. Giannikis* G. Alonso* D. Fauser† D. Kossmann*

*Systems Group, Department of Computer Science, ETH Zurich, Switzerland

†Amadeus IT Group SA, France

ABSTRACT

This paper introduces Crescendo: a scalable, distributed relational table implementation designed to perform large numbers of queries and updates with guaranteed access latency and data freshness. To this end, Crescendo leverages a number of modern query processing techniques and hardware trends. Specifically, Crescendo is based on parallel, collaborative scans in main memory and so-called “query-data” joins known from data-stream processing. While the proposed approach is not always optimal for a given workload, it provides latency and freshness guarantees for *all* workloads. Thus, Crescendo is particularly attractive if the workload is unknown, changing, or involves many different queries. This paper describes the design, algorithms, and implementation of a Crescendo storage node, and assesses its performance on modern multi-core hardware.

1. INTRODUCTION

In the last decade, the requirements faced by database applications have changed significantly. Most importantly, databases must operate with predictable performance and low administration cost. Furthermore, databases must be able to handle diverse, evolving workloads as applications are constantly extended with new functionality and new data services are deployed, thereby adding new types of queries to the workload in an unpredictable way. Most notably, these new requirements have been expressed in the context of platforms such as eBay, Amazon, Salesforce, etc. Salesforce, for instance, allows users to customize their application and define their own queries. Providing such a platform involves highly diverse query workloads; yet, users of the platform expect a constant response time.

Unfortunately, throughput and latency guarantees are difficult to make with traditional database systems. These systems are designed to achieve *best* performance for every individual query. To this end, they rely on sophisticated query optimizers and skilled administrators for selecting the right indexes and materialized views. Such complex systems are expensive to maintain and do not exhibit predictable performance for unpredictable, evolving workloads.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, to post on servers or to redistribute to lists, requires a fee and/or special permission from the publisher, ACM.

VLDB '09, August 24-28, 2009, Lyon, France

Copyright 2009 VLDB Endowment, ACM 000-0-00000-000-0/00/00.

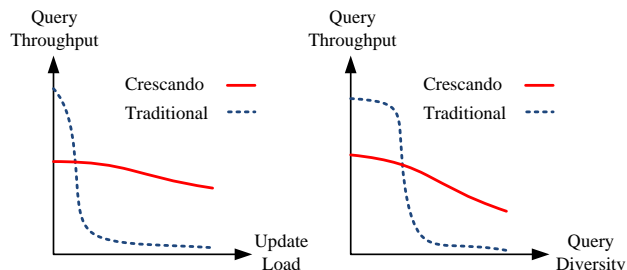


Figure 1: Crescendo vs. Traditional Databases

As a solution to this problem, we present a novel relational table implementation, Crescendo, which offers significantly higher predictability than traditional databases. Consider Figure 1. It sketches two charts that compare the desired behavior of Crescendo to that of a traditional database. The performance study in this paper includes an actual experiment and chart (Fig. 11) of this kind.

If the update load is light, then a traditional database can support high query throughput by offering just the right indexes and materialized views to support the queries. Unfortunately, the query throughput drops quickly with an increasing update load. Likewise, as shown in the second chart of Figure 1, the query throughput decreases rapidly with the number of different query types, as more and more queries require full-table scans. The effects shown in Figure 1 compound, resulting in even lower throughput for workloads with high query diversity *and* concurrent updates.

Crescendo tables are designed for exactly such workloads. Crescendo tables may be inferior to traditional solutions for their sweet spot, but they exhibit good and, more importantly, *predictably* good performance for *all* workloads. Crescendo achieves this by combining and extending a number of database techniques, some of which have been explored recently [18, 17, 25] for data warehousing:

1. Crescendo is based on a *scan-only* architecture (i.e., no indexes) in order to achieve predictable performance.
2. Crescendo uses main-memory storage and data partitioning to scale-up linearly on multi-core machines.
3. Crescendo employs collaborative (shared) scans, in order to overcome the memory-bandwidth bottleneck.

Crescendo features a novel collaborative-scan algorithm, called Clock Scan, to achieve both high query and update throughput with predictable latency. The idea behind the Clock Scan algorithm is to batch incoming queries, and model query/update processing as a *join* between queries and update statements on the one side, and the table on

the other side. For main-memory databases, index nested-loop joins are particularly effective, because random access is cheap. But rather than indexing the table, as done in traditional databases, Crescando indexes the *queries*, as proposed for publish/subscribe systems [6, 9]. Crescando introduces efficient join algorithms which support not only queries, but also updates. We refer to the latter as an *update-data join*.

In summary, this paper makes the following contributions:

- a novel cooperative scan algorithm, Clock Scan;
- the first update-data join algorithm, Index Union Update Join, which ensures predictable query throughput and latency under a heavy, concurrent update load;
- an efficient recovery scheme for scan-based in-memory query processing;
- a comprehensive performance evaluation of Crescando, which integrates Clock Scan, Index Union Update Join, and the new recovery scheme.

The remainder of this paper is organized as follows: Section 2 gives a real life use case that motivated this work and derives quantified requirements from it. Section 3 presents the architecture of Crescando. Section 4 gives details of the Clock Scan algorithm. Section 5 presents query/update-data join algorithms used by Clock Scan. Section 6 summarizes the benefits of memory partitioning (*segmentation*). Section 7 discusses transactional properties and Crescando’s recovery scheme. Section 8 shows the results of an extensive performance evaluation. Section 9 discusses related work. Section 10 concludes the paper with a selection of avenues for future work.

2. PROBLEM STATEMENT

This section presents the real-world application scenario (airline reservation systems) that motivated the design of Crescando. Furthermore, it specifies again the particular data processing requirements that Crescando addresses.

2.1 Use Case

Amadeus is a world-leading service provider for managing travel-related bookings (flights, hotels, rental cars, etc.). Its core service is the Global Distribution System (GDS), an electronic marketplace that forms the backbone of the travel industry. The world’s largest airline carriers and many thousand travel agencies use the GDS to integrate their data.

The core database in the Amadeus GDS contains dozens of millions of flight bookings. For historical and performance reasons, the authoritative copy of each booking is stored in a denormalized BLOB (binary large object) of a few kilobytes, directly accessible through a unique key. For the bookings that need to be kept on-line, this results in a single, flat fact table of several hundred gigabytes in size. This BLOB table currently sustains a workload of several hundred updates and several thousand key-value look-ups per second.

Key-value access is sufficient for all transactional workloads faced by the system. However, it is ill-suited to answer the increasing amount of real-time, decision-support queries that select on non-key attributes, for example: “give the number of first class passengers in a wheelchair, who depart from Tokyo to a destination in the US tomorrow.” Queries like this are increasingly common and feature stringent latency constraints, because operational decisions are made based on their results.

To support such queries, Amadeus maintains a growing number of materialized relational views on the BLOB table, some of which are updated in real-time through a proprietary event streaming architecture. The very existence of these materialized views implies that there are few joins in the workload. The vast majority of queries are of the form `SELECT <Attr1>, <Attr2> ... FROM <View> WHERE ...`, with occasional aggregation.

The largest existing view is a denormalization of flight bookings: one record for every person on a plane. This is the view used for the performance evaluation in this paper, and we will refer to it as *Ticket* in this context. A Ticket record is approximately 350 bytes in size (fixed), and consists of 47 attributes, many of which are flags with high selectivity (e.g., seat class, wheelchair, vegetarian). Since one travel booking may be related to multiple persons and flights, Ticket contains hundreds of millions of such records.

Ticket is updated a few hundred times per second, in real-time. Update rates may be many times higher for brief periods, as bad weather or security incidents can cause large bursts of passenger reaccommodation requests. The update load is increasing at a lower rate than the query load, but is already causing severe problems with regard to index maintenance in the current setup.

The view is used in a large number of data services: from generating the passenger list of a single flight to analyzing the customer profile of different airlines and markets (pairs of <source, destination> airports). Since the system has reached a level of complexity where adding views and indexes is no longer feasible let alone economical, a growing number of queries on Ticket do not match the primary index on <flight number, departure date>.

As a result, more and more queries have to be answered in batch (off-line) using full-table scans, with a dramatic impact on performance during this period. Other queries which do not match the index and do not allow for batch-processing are simply not allowed. As a solution to all these queries that do not warrant a view of their own, we propose a single instance of Ticket based on Crescando.

2.2 Requirements

We quantify the requirements of the Amadeus use case as follows:

Query Latency Any query must be answered within two seconds.

Data Freshness Any update must be applied and made visible within two seconds.

Query Diversity The system must support any query, regardless of its selection predicates.

Update Load The average load is 1 update/GB*sec. Peak load is 20 updates/GB*sec for up to 30 seconds.

Scalability The system must scale linearly with the read workload by adding machines (scale-out) and CPU cores to individual machines (scale-up).

Our design goal is to maximize query throughput per CPU core under those constraints.

2.3 Other Use Cases

The Amadeus use case represents just one deployment scenario for Crescando. As a general relational table implementation, it is equally possible to use Crescando for base tables, or build hybrid systems that combine traditional in-

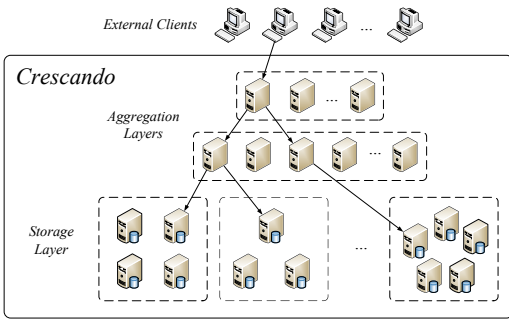


Figure 2: Distributed Architecture Overview

dexed tables with Crescando tables and perform joins and complex aggregation on top of Crescando.

Many systems in real-time business intelligence and decision support face requirements similar to those of Amadeus. We do not see Crescando as a replacement to all existing database technology, but as a complement that widens the range of requirements that can be met.

3. ARCHITECTURE AND FRAMEWORK

This section gives an overview of the architecture of Crescando, a relational table implementation. Because main memory is limited, a single machine might be unable to store the entire table. So for scalability and availability, we propose a distributed architecture based on horizontal data partitioning and replication. This paper focuses on the internals of a *single*, non-replicated storage node. Nonetheless, we give an outlook on the distributed architecture, also because it provides a strong argument in favor of indexing queries rather than data.

3.1 Distributed Architecture

Crescando horizontally partitions the table between *replication groups*, which consist of *storage nodes*, whose data is accessed in read-one write-all (ROWA) fashion through *operations*. The replication groups form the *storage layer*. An operation is either a *query* (simple SQL-style `SELECT` statement with optional scalar aggregation) or an *update*. In this paper, we use the term update for any unnested, SQL-style `INSERT`, `UPDATE`, or `DELETE` statement. We write `UPDATE`, `INSERT`, or `DELETE` whenever we want to make a distinction.

One or more layers of *aggregator nodes* are responsible for routing operations to replication groups, and merging (“aggregating”) the results. Figure 2 visualizes the distributed architecture. It is similar to that of the NDB storage engine used by MySQL Cluster [19], to name just one example.

In traditional architectures, administrators tune performance by providing special views or indexes on one storage node but not on others, or by using entirely different technology for certain replicas (*heterogeneous replication*). In Crescando, these tuning knobs do not exist. Replicas are completely homogeneous.

Still, clustering queries based on their selection predicates is beneficial to performance. For example, assume all queries with a selection predicate on flight number go to replica A, while all queries with a selection predicate on airport go to replica B. Scale-up is potentially *super-linear* because similar queries can be indexed and processed together very efficiently, as shown in detail later in the paper (Section 5).

In Crescando, clustering decisions are made autonomi-

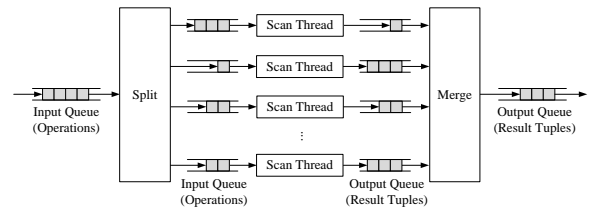


Figure 3: Storage Node Architecture Overview

cally, at *runtime*. Query indexes are extremely short-lived, so the query clustering can change at any time. In a traditional architecture, losing a special view or index has a dramatic impact on at least part of the workload. In contrast to this, losing a replica in Crescando causes throughput to decrease by roughly the same, predictable degree for all queries. This enables predictable performance in high-availability setups without additional data redundancy.

3.2 Storage Node Architecture

At the time of writing, we have fully implemented the Crescando storage node, with the aggregation node being under development. Figure 3 visualizes the storage node architecture. Storage nodes expose two main functions: enqueue an operation, and dequeue a result tuple. Rather than enqueueing an operation and waiting for the result, the users (i.e., aggregator nodes) are expected to concurrently enqueue a large number of operations and to asynchronously dequeue results. Each aggregator node in turn may serve thousands of external clients.

Once inside a storage node, an operation is split and put into the input queue of one or more scan threads. Each scan thread is a kernel thread with hard processor affinity, which *continuously* scans a horizontal partition of the data, stored in a dedicated partition of memory we call a *segment*.

Scan threads periodically remove operations from their input queue and *activate* them. At any given moment, a scan thread may have multiple active operations. As a scan thread executes its set of active operations against the records under the scan cursor, it generates a stream of result tuples. Once an operation has completed a full scan of a data partition, the scan thread puts a special end-of-stream tuple on the output queue and deactivates the operation.

The architecture raises questions with regard to fairness (“cheap” versus “expensive” queries) and resource utilization (busy versus idle threads). For one thing, the fact that every operation takes roughly the same time is a key feature and a strong type of fairness. For another thing, Crescando relies on the law of big numbers. The more operations share a scan cursor, the more they are representative of the workload as a whole, thereby balancing the load across scan threads. The algorithms introduced in this paper allow *thousands* of operations to share a scan cursor.

At this point, Crescando uses the traditional N-ary Storage Model (NSM), also known as *row-storage*. We are aware of alternative storage models which may improve cache locality, namely Partition Attributes Across (PAX) and the Domain Storage Model (DSM) [1]. These techniques are complementary to our approach. Having that said, our experimental results show that memory bandwidth is not a bottleneck in Crescando. The algorithms we introduce are clearly CPU bound under load, making vertical partitioning much less interesting than in traditional query processing.

Algorithm 1: Classic Scan Thread

```
Data: Segment seg
Data: OpQueue iq; // input query and update queue
Data: ResultQueue oq; // output queue
while true do
  Op op ← iq.get(); // activate a single operation
  // scan the full segment, slot-wise
  foreach Slot s ∈ seg do Execute(op, s, oq)
  Put(oq, EndOfStream(op)); // deactivate the operation
```

Algorithm 2: Elevator Scan Thread

```
Data: Segment seg
Data: OpQueue iq; // input query and update queue
Data: OpQueue aq; // active query and update queue
Data: ResultQueue oq; // output queue
while true do
  // scan the full segment, slot-wise
  foreach Slot s ∈ seg do
    // execute all active operations against the slot
    foreach Op op ∈ aq do Execute(op, s, oq)
    // deactivate all operations that finished a full scan
    while Finished(Peek(aq)) do
      Put(oq, EndOfStream(Get(aq)))
    // activate all operations in the input queue
    while ¬IsEmpty(iq) do Put(aq, Get(iq))
```

4. SCAN ALGORITHMS

A main contribution of this work is the development of a new scan algorithm, Clock Scan, which models query/update processing as a join between a set of queries/updates and a table. This section introduces Clock Scan and compares it to the state of the art, Classic and Elevator Scan.

All scan algorithms *continuously* scan the data in a separate thread of control. Also, the algorithms operate on *slots* of fixed-sized records. Extending Crescendo to variable-sized records or different record layouts would affect the algorithms to some degree, but poses no conceptual problems, since there are no auxiliary data structures (indexes), and records can be arranged freely.

4.1 Classic Scan

In a straight-forward, *Classic* implementation of the split-scan-merge pipeline (Fig. 3), each scan thread processes one incoming operation at a time. This first, naïve variant is shown in Algorithm 1.

The `Execute` function of an operation first checks whether the slot is occupied. If it is not, and the operation is an `INSERT`, a record is inserted into the slot (“first fit” policy). If the slot is occupied, and the operation is not an `INSERT`, the operation’s selection predicates are evaluated. If all predicates are satisfied, the function either puts a result tuple on the output queue (`SELECT` operation), or modifies the slot (`UPDATE`, `DELETE`). After processing all records, Classic Scan puts a special end-of-stream tuple on the output queue and gets the next operation from the input queue.

The asymptotic runtime of Classic Scan is $O(n * m)$ for n operations over m slots. Obviously, it takes little advantage of the computational resources of modern processors, as it makes essentially no use of the cache.

4.2 Elevator Scan

A first improvement over Classic Scan is Elevator Scan. Zukowski et. al. [25] and Raman et. al. [18] have previously investigated variants of Elevator Scan for read-only

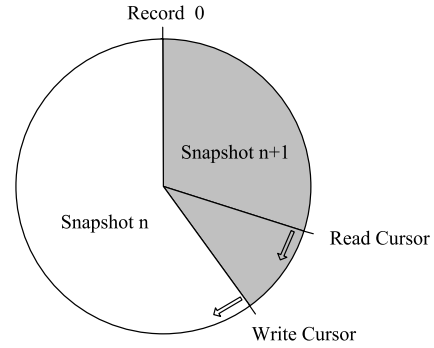


Figure 4: Clock Scan Idea

workloads in disk-based and main-memory databases respectively. Algorithm 2 shows our generalization of Elevator Scan for mixed workloads.

Elevator Scan maintains a queue of active operations *aq*, which are executed, in arrival order, against the slot under the scan cursor before moving on to the next slot. Executing operations strictly in arrival order guarantees a high degree of consistency even if some operations are writes.

Algorithm 2 updates the active queue at every slot. All active operations which have finished a full scan are deactivated, and the input queue is flushed. Our concrete implementation does this only at *chunk* boundaries (equivalent to pages in a disk-based database). Also, our implementation splits the active queue into multiple queues of different type, to avoid executing `DELETES` on an empty slot, for example.

Elevator Scan is a so-called *cooperative* scan, in that it lets multiple operations share the scan cursor to improve cache locality and overcome the infamous *memory wall* [24, 4]. However, the asymptotic runtime of Elevator Scan is still $O(n * m)$ for n operations over m slots.

4.3 Clock Scan

Even though Elevator Scan greatly improves upon the cache behavior of Classic Scan, this improvement is at most a constant factor in runtime. In contrast, Clock Scan performs query/update-data joins over sets of queries/updates to allow *asymptotic* runtime improvements. In this section, we are chiefly concerned with the scan algorithm itself. Query/update-data joins are covered in detail in Section 5.

Figure 4 shows a high-level illustration of the algorithm idea. Suppose we continuously run two circular scans over the segment: one read scan, one write scan. Let us enforce that the read cursor cannot pass the write cursor and vice versa, i.e., the read cursor is always some delta less than one cycle behind the write cursor. The write cursor executes updates strictly in arrival order. It can be proven easily that the read cursor will always see a consistent snapshot if the algorithm only activates operations at record 0, regardless of the *order* in which queries are executed.

Clock Scan, given in Algorithm 3, merges the two logical cursors into a single physical cursor for higher cache locality. At each iteration of the infinite loop, it first flushes the input queues and creates join plans for the active queries and updates (cf. Section 5.4). Then it performs the actual, chunk-wise scan of the segment, joining each chunk of records with the set of queries and the set of updates.

The runtime complexity of Clock Scan is determined by the joins. Clock Scan is correct if the join algorithms are correct (cf. Section 5.1). In particular, update joins must

Algorithm 3: Clock Scan Thread

```
Data: MultiQueryOptimizer opt
Data: Segment seg
Data: OpQueue iqq, iuq; // input query and update queues
Data: ResultQueue oq; // output queue
while true do
  //activate all updates in input update queue
  UpdateSet us  $\leftarrow \emptyset$ 
  while  $\neg$ IsEmpty(iuq) do Put(us, Get(iuq))
  //activate all queries in input query queue
  QuerySet qs  $\leftarrow \emptyset$ 
  while  $\neg$ IsEmpty(iqq) do Put(qs, Get(iqq))
  //do multi-query optimization
  UpdatePlan up  $\leftarrow$  PlanUpdates(opt, us)
  QueryPlan qp  $\leftarrow$  PlanQueries(opt, qs)
  //scan the full segment, chunk-wise
  foreach Chunk c  $\in$  seg do
    Join(up, c, oq); //update-data join
    Join(qp, c, oq); //query-data join
  //deactivate all active operations
  foreach Op op  $\in$  qs  $\cup$  us do Put(oq, EndOfStream(op))
```

leave the data relation in the same state as executing the set of updates in serialization (activation) order.

5. QUERY-DATA JOINS

Clock Scan allows asymptotically better performance than Elevator Scan because it reorders and interleaves queries to perform query/update-data joins. The term *query-data join* has been coined by Chandrasekaran et al.[6] and is based on the idea of interpreting a set of pending queries as a relation of predicates. In this section, we first give a semi-formalization of the idea, before introducing two concrete join algorithms and a multi-query optimizer for planning these joins.

5.1 Queries as a Relation

In the following introduction to query-data joins, we will be concerned with a set of queries over a single data relation R . We restrict ourselves to those queries 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.

Now, let us consider a set of queries, each with a single equality predicate of the form $attrib = const$, for example $age = 12$. We can define a relation Q with heading $(qid, const)$, where qid is the unique query-id and $const$ is the constant to test equality with. The union of the results of these queries is exactly the join $Q \bowtie_{attrib=const} R$.

One can extend this model in a straight-forward fashion to range predicates of the form $lb \leq attrib \leq ub$ where lb denotes the lower-bound constant, and ub denotes the upper-bound constant. We define the relation Q as (qid, lb, ub) and perform the band join $Q \bowtie_{lb \leq attrib \leq ub} R$. Note, that $\bowtie_{attrib=const}$ is a special case of $\bowtie_{lb \leq attrib \leq ub}$ where $lb = ub$, so queries which test equality can participate in band joins.

If queries have other predicates, these can either be tested on the result of the join, or become part of the join by extending Q and the join predicate. Queries which do not have a predicate on a specific attribute $attrib$ can be modeled as having an unbounded range predicate on $attrib$.

Algorithm 4: Index Union Join

```
Input: Chunk c
Input: IndexSet is; // predicate indexes
Input: QuerySet qs; // unindexed queries
Input: ResultQueue oq; // output queue
foreach Record r  $\in$  c do
  //probe the indexes for candidates
  foreach Index i  $\in$  is do
    QuerySet C  $\leftarrow$  Probe(i, r); //candidate queries
    foreach Query q  $\in$  C do Execute(q, r, oq)
  //execute unindexed queries
  foreach Query q  $\in$  qs do Execute(q, r, oq)
```

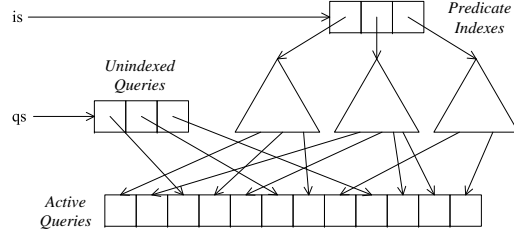


Figure 5: Data Structures for Index Union Join

The idea of query-data joins has to this point only been investigated in the context of data stream processing and publish-subscribe systems, where sharing work between a set of continuous queries is a main topic of interest [9, 6].

In this paper, we introduce a new, related concept we call *update join*. An update join interprets the selection predicates of a sequence of updates as an ordered relation, and leaves the data relation in the same state as executing the set of updates serially, in order.

5.2 Index Union Join

In the course of our work, we have experimented with a number of existing join algorithms, such as a partitioned sort-merge band join [8] for large range predicates. Such “state-heavy” join operators turned out to be only effective when all queries shared a single predicate attribute, in which case one might as well index the data. Performance degraded quickly for multi-attribute join predicates, as most tuples in the query relation became infinite ranges.

A more flexible and general solution is to *index* predicates, an idea inspired by publish-subscribe systems such as Le Subscribe [9]. We have designed and implemented a cache-conscious query-data join based on short-lived predicate indexes: *Index Union Join*.

A predicate index maps a single attribute value to a set of queries, as defined in section 5.1. For example, the three queries $q_1 : age = 12$, $q_2 : age = 27$, and $q_3 : age = 27$ could be indexed in a multi-hash map that returns $\{q_2, q_3\}$ when probed with a record with $age = 27$. Range predicates such as $30 < age < 50$ can be indexed in any spatial index structure that supports stab queries.

Figure 5 visualizes the data structures of Index Union Join. There exists exactly one access path to each query. Either one of the query’s predicates is part of a predicate index, or the query is part of the set of unindexed queries. Since this constitutes a partitioning of the set of queries, following every access path (every index plus the set of unindexed queries) with every record and unioning the result tuples yields the same result relation as executing every query against every record. Algorithm 4 follows straight from this

Algorithm 5: Index Union Update Join

```
Input: Chunk  $c$ 
Input: IndexSet  $is$ ; // predicate indexes
Input: UpdateSet  $us$ ; // unindexed UPDATES, DELETES
Input: InsertQueue  $iq$ ; // INSERTs
Input: ResultQueue  $oq$ ; // output queue
foreach Slot  $s \in c$  do
  Timestamp  $t \leftarrow 0$ 
  while  $t < \infty$  do
    if  $IsOccupied(s)$  then
      //slot is occupied; perform updates
       $t \leftarrow PerfUpdates(s, t, is, us)$ 
    else if  $\neg IsEmpty(iq)$  then
      //slot not occupied, have insert; execute it
       $Insert\ i \leftarrow Get(iq)$ 
       $Execute(i, s)$ 
       $t \leftarrow i.timestamp + 1$ 
    else  $t \leftarrow \infty$ ; //not occupied, no insert; next slot
```

observation. Note that putting all the results of executing queries into a common output queue gives union semantics.

The worst-case runtime complexity of Index Union Join (every record matches every index) is no better than executing every query against every record, as done by Classic Scan and Elevator Scan. However, Index Union Join is faster for any reasonably selective set of predicates, because probing an index immediately takes all non-matching queries out of consideration. Runtime is dominated by the cost of probing the index, which is constant or logarithmic. This is analogous to the difference between a nested-loop join and an index nested-loop join in traditional query processing. For large sets of queries, the resulting speed-up is significant, as shown in Section 8.

Different to the given pseudo-code, our optimized implementation vectorizes the index probing, i.e., it passes the entire chunk to an index’ *probe* function instead of a single record. This gives significantly higher data and instruction cache locality. Also, the implementation uses a visitor pattern to execute all matching queries inside the *probe* function of the index instead of using a candidate query set C .

In terms of indexes, Crescendo currently implements a jagged array for attributes with small domain (e.g. gender), a chained hash index with linear probing for equality predicates, as well as a packed 1-dimensional R-Tree [11] for range predicates. These simple index structures turned out to have better performance than more complex structures we experimented with, due to their high data and instruction cache locality.

5.3 Index Union Update Join

Under a heavy update load, one would like to use predicate indexes also for updates. The problem is that updates have to be executed in serialization order, which we will express as timestamps. What makes this hard to do efficiently is the fact that a slot’s state may change after each update, thereby changing the set of matching updates.

Index Union Update Join given in Algorithm 5 solves the problem. It maintains a queue iq of (unindexable) INSERTs, and a set of predicate indexes is , while us contains unindexed UPDATES and DELETES. For simplicity, we assume that each INSERT consists of exactly one record. The implementation of the algorithm does not have this restriction.

The function `PerfUpdates` is an extension of the Index Union Join shown in the previous section. It collects a set

Function `PerfUpdates`(Slot s , Timestamp t , IndexSet is , UpdateSet us): Timestamp

```
UpdateSet  $C \leftarrow us$ ; //candidate set
UpdateSet  $M \leftarrow \emptyset$ ; //match set
//probe indexes for additional candidates
foreach Index  $i \in is$  do  $C \leftarrow C \cup Probe(i, s.record)$ 
//find matches among candidates
foreach Update  $u \in C$  do
  if  $Matches(u, s.record)$  then  $M \leftarrow M \cup \{u\}$ 
if  $M \neq \emptyset$  then
  //execute match with lowest timestamp
  Update  $u \leftarrow \min_{u.timestamp} \{u \in M\}$ 
  if  $u.timestamp \geq t$  then
     $Execute(u, s)$ 
    if  $IsDelete(u)$  then
      return  $u.timestamp + 1$ ; //slot empty; return
    else
      //slot updated; recurse
      return  $PerfUpdates(s, u.timestamp + 1, is, us)$ 
return  $\infty$ 
```

M of all updates matching $s.record$. Then, it looks for the update $u \in M$ with the lowest timestamp greater or equal to t (if any) and executes it. The variable t is initially 0 and ensures updates are executed in timestamp order as follows. If u was a DELETE, recursion ends (the slot is empty). Otherwise, the function recurses for $t = u.timestamp + 1$. This ensures that no update v where $v.timestamp \geq u.timestamp$ will be executed on the updated record, even though v remains in the indexes and may repeatedly appear in M as the function recurses.

Index Union Update Join proceeds to the next slot when t becomes ∞ (some value greater than any legal timestamp). This happens if and only if the slot is occupied but no matching update with a timestamp $\leq t$ exists, or the slot is empty but no insert operations remain in iq . A formal proof of correctness is not difficult but omitted for space reasons.

As for performance, we note that in the *worst-case*, there are only UPDATES to be joined, and each of those n UPDATES matches every record, every time `PerfUpdates` is called. Since $|M| \leq n$, the depth of the recursion is up to n and the worst-case runtime complexity for m records is $O(n^2 * m)$. In reality, of course, M will typically contain 0 or 1 updates, so runtime is dominated by the cost of probing the indexes, which is constant or logarithmic in n . In our optimized implementation, the “recursion” is just an assignment to t and a goto-statement. Also, the candidate set C is replaced by a visitor pattern as with Index Union Join.

5.4 Multi-Query Optimization

The term *multi-query optimization* traditionally refers to the practice of finding common sub-expressions among a set of queries, with the goal of sharing and pipelining intermediate results (partial selections and joins) in a global (multi-) query plan [22, 12]. Such an approach is useful for small sets of long-running, complex queries.

In contrast to this, Crescendo is designed for large sets of short-running, simple queries. The optimization problem we are interested in is finding a set of predicate indexes which minimize the cost (runtime) of the join algorithms given previously. Also, when we talk about multi-query optimization here, we really mean queries and updates, since they are indexed in exactly the same way. In any case, the problem is NP-hard, as the special case “find the minimum set of

Algorithm 7: Multi-Query-Optimizer

Data: Gain *thresh*; // minimum gain threshold
Input: OpSet *os*; // active queries/updates
Input: AttributeSet *A*; // indexable attributes in schema
Output: IndexSet *is* $\leftarrow \emptyset$; // predicate indexes
QuerySet *uos* $\leftarrow os$; // unindexed queries/updates
repeat
 Attribute *a* $\leftarrow \max_{\text{Gain}(a)} \{a \in A\}$
 Gain *g* $\leftarrow \text{Gain}(a)$
 if *g* $\geq \text{thresh}$ **then**
 Index *idx* $\leftarrow \text{BuildIndex}(a, uos)$
 is $\leftarrow is \cup idx$
 uos $\leftarrow uos \setminus \{q \in idx\}$
 A $\leftarrow A \setminus \{a\}$
until *g* $< \text{thresh}$

predicate indexes to cover all queries/updates” is already an instance of minimum set covering.

Given the short lifetime of a query/update plan (about 1 second), finding an optimal solution is out of the question. So instead, the optimizer uses a greedy algorithm we developed, shown in Algorithm 7. At each iteration, it builds an index on the attribute that gives the highest *gain*, and then takes all queries/updates that are covered by the index out of consideration. The **Gain** function is defined as:

$$\text{gain}(Q, a) := \sum_{q \in Q} 1 - \text{selectivity}(a, q)$$

It is based on the following idea. Touching a query/update *q* is associated with a certain overhead (branching, accessing attribute values etc.) one wants to avoid. The probability of touching *q* after probing a predicate-index on *a* is *q*’s selectivity with respect to *a*. If *q* does not have a predicate on *a*, that probability is 1. The gain is then the expected number of operations in *Q* that do *not* have to be touched given an index on *a*. Obviously, maximizing this number minimizes the number of operations expected to be touched. The empirically obtained value *thresh* (currently 2.5) prevents the optimizer from building more indexes than beneficial.

At this point, the optimizer does not weigh in the space and CPU cost of alternative index types (e.g. hash versus tree, integer versus string). In combination with out-of-order activation of queries, this forms a challenging multi-dimensional optimization problem. More complex gain metrics and optimizer heuristics that take these variables into account represent an interesting extension point of our work.

To compute the gain metric and pick a suitable set of predicate indexes, the optimizer requires an estimate of the selectivity of each predicate. Crescando keeps a small set of statistics that can be efficiently maintained by just scanning the data: number of records, number of null values for each attribute, and number of distinct values for each attribute.

For computing these statistics, Crescando employs a simple yet effective technique known as *linear* or *probabilistic counting* [10, 23]. The probabilistic counting algorithm is implemented as a side-effect of a periodical *statistics query*, which to the scan algorithm is just an unconditional query.

6. SEGMENTATION

As outlined in Section 3, Crescando partitions the physical memory into disjunct (non-overlapping) segments and assigns them to dedicated processor cores. We refer to this technique as *segmentation*. Each core runs a single scan thread with hard affinity (threads do not migrate between

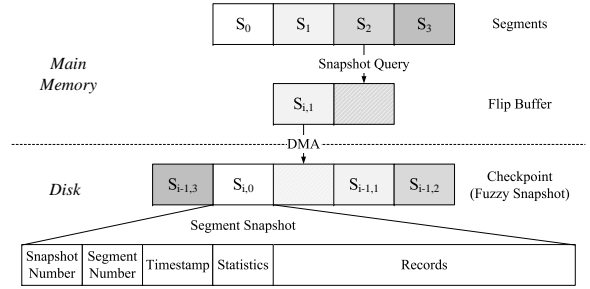


Figure 6: Checkpointing Overview

processors). This shared-nothing architecture enables linear scale-up because of the following key properties:

No Locking Because a scan thread is guaranteed to be the only one updating records in its segment, execution can proceed without any locks or latches.

Maximum Cache Coherency Distinct processor caches never contain a copy of the same record, so they are implicitly coherent in this performance-critical respect. Records need not be written back to main memory until the scan cursor moves on, even if they are modified.

Minimum NUMA Distance Because scan threads have hard processor affinity, their respective memory segments are uniquely associated with a processor. This is critical on NUMA (non-uniform memory access) architectures such as AMD’s Opteron processors or Intel’s Nehalem architecture. Using segmentation, CPUs never access each other’s local memory except for passing operations and operation results, giving maximum memory bandwidth and minimum access latency where it matters: predicate evaluation.

7. TRANSACTIONAL PROPERTIES

Transactions can be implemented in Crescando in almost the same way as in any other database system. In particular, atomicity and checking integrity constraints are orthogonal to the design of Crescando. More care needs to be taken with regard to durability and isolation.

For durability, Crescando uses a combination of write-ahead logging and checkpointing. At any time, Crescando maintains a checkpoint on disk in the form of a *fuzzy snapshot*, i.e., a set of timestamped snapshots of each memory segment, which are obtained through unconditional *snapshot queries*. When executed against a record, snapshot queries copy the record to a flip buffer which asynchronously writes the records back to disk.

Consider Figure 6 for an illustration. By scheduling snapshot queries on segments in a round-robin fashion, $n + 1$ segments of disk space are sufficient to always have a checkpoint of n segments of memory. The constant-size flip buffer ensures that a slow disk does not block a snapshot query, i.e., the scan thread executing that query.

After a crash, recovery proceeds in parallel for each segment. First, the snapshot segment is loaded from disk, then the log is replayed. Currently, Crescando implements a logical redo-log, which is extremely efficient, but also implies an auto-commit model. An extension to full atomicity is straight-forward, by adding a physical log and an undo-phase to log replay, as known from ARIES [16].

As for the overhead of snapshot selects, our experiments have shown that the access latency of the snapshotted segment increases exactly by the time it takes to perform a `memcpy` of its contents. Under a heavy load, when scans take a multiple of this time, this becomes negligible. Despite the low overhead, logging and checkpointing by every node may be overkill in a highly replicated setup (cf. Section 3.1), and recovery is a strictly optional feature. For space reasons we omit dedicated experiments on logging and recovery from this paper. All the experiments in Section 8 were run with logging turned off for easier interpretation and better comparability to existing technology.

With regard to isolation, the design of Crescando favors optimistic and multi-version concurrency control. For instance, snapshot isolation [3] can easily be implemented by keeping old versions of updated records. Locking can also be implemented, but since Crescando does not maintain any indexes, it is difficult to implement range locks [15].

For the performance experiments reported in Section 8, we used a simplified transaction model in which every query runs in a separate transaction. For this transaction model, Crescando implicitly supports write monotonicity and serializability. This simplified transaction model is sufficient for the use case described in Section 2.

8. PERFORMANCE EVALUATION

The goal of our work is predictable performance for unpredictable workloads. Crescando seeks to achieve this by providing a high degree of *robustness* in latency to query diversity, query volume, and concurrent updates. Crescando is also expected to scale linearly on modern multi-core hardware. This section presents the results of a performance evaluation which covers both these aspects (Part I), and compares Crescando to a traditional main-memory database, MySQL 5.1 and its *Memory* storage engine (Part II).

8.1 Implementation Notes

The main component of Crescando storage nodes is the storage engine. Other components include networking and configuration. We have implemented the storage engine as a shared library for 64bit POSIX systems, written in highly optimized C++ with a few lines of inline assembly. Similar to other embedded and main-memory databases, a small fraction of schema-dependent code is generated by a schema compiler and loaded at runtime.

The engine offers a simple C interface with two main functions: enqueue an operation, and dequeue a result. Once inside the engine, its *controller* module is in charge of forwarding the operation to one or more scan threads, and later merging the results. The controller chooses where to forward operations by means of a pluggable *segmentation strategy*. In this paper, we present results for two simple strategies: round-robin and hash partitioning.

Our implementation includes all the algorithms presented in this paper. Clock Scan uses the multi-query optimizer and query-data join algorithms described in Section 5. Each scan runs in a dedicated kernel thread with hard processor affinity. The engine makes sure that memory segments are local to the corresponding scan thread by allocating all memory through *libnuma*, a NUMA API for Linux [13]. To maximize cache locality and minimize branching, the implementation makes heavy use of function inlining, template metaprogramming, and uses vectorization and other low-level op-

Algorithm 8: Attribute Pick Algorithm

```

Input:  $N, D, s$ 
Output:  $P$ 
 $Z \leftarrow \text{Zipf}(s, N);$  //initialize  $Z$  as Zipf distribution
 $V \sim \mathcal{B}(N, 1/D);$  //get random  $V$  acc. to binomial dist.  $B$ 
for  $v = 1$  to  $V$  do
     $a \sim Z;$  //get random  $a$  according to  $Z$ 
     $p \leftarrow Z[a];$  //get probability  $p$  of  $a$  according to  $Z$ 
     $Z[a] \leftarrow 0;$  //remove  $a$  from  $Z$ 
     $Z \leftarrow Z/(1-p);$  //re-normalize remaining  $Z$ 
     $P \leftarrow P \cup a;$  //add  $a$  to result set  $P$ 

```

timizations described by Boncz et al.[4, 5] and Ross[20, 21].

8.2 Test Platform

We conducted all experiments on a 16-way machine built from 4 quad-core AMD Opteron 8354 (“Barcelona”) processors with 32 GB of DDR2 667 RAM, for a cumulative memory bandwidth of over 42 GB/sec. Each core had a 2.2 GHz clock frequency, 64 KB + 64 KB (data + instruction) L1 cache, and 512 KB L2 cache. The machine was running a 64-bit Linux SMP kernel, version 2.6.27.

8.3 Workload

We recreated the Ticket schema introduced in Section 2.1, and ran all experiments on Amadeus’ live data. For the experiments we generated two types of workload. The first was a representative subset of the real-world queries and updates of Amadeus with their relative frequencies. The second was a synthetic workload with variable predicate attribute skew.

8.3.1 Amadeus Workload

At the time of writing, the average, real query against the Ticket table has 8.5 distinct predicate attributes and fetches 27 of the 47 attributes in the schema. 99.5% of the queries feature an equality predicate on <flight number, departure date>. `UPDATES` typically affect just a few records, and always feature equality predicates on booking number. `INSERTS` and `DELETES` occur at about 1/10th the rate of `UPDATES`. `DELETES` typically affect specific records just like `UPDATES`.

8.3.2 Synthetic Workload

As motivated in the introduction, we were not as much interested in the performance under the current, index-aware workload, as in the performance under future, increasingly diverse and unpredictable workloads; i.e., we were interested in workloads where users ask *any query they want*, uninhibited by knowledge of the table’s physical organization.

For this purpose, we created a synthetic workload as follows. We kept all the updates of the Amadeus workload (their composition is very stable over time), and replaced the real queries by a synthetic set of queries with a particular, *skewed* predicate attribute distribution. To create such a skewed set of predicate attributes for a synthetic query, we invoked Algorithm 8.

Executing the algorithm with $N = 47$ (total attributes in schema), $D = 9$ (average predicates per query), and variable shape parameter s yields the attribute frequencies given in Figure 7. s is the characteristic exponent of the Zipf distribution used by Algorithm 8 to initialize the probability function Z . The figure includes the attribute frequencies of the current Amadeus workload for comparison. Notice that the area under each synthetic curve is $D = 9$, as desired. The area under Amadeus’ curve is roughly 8.5.

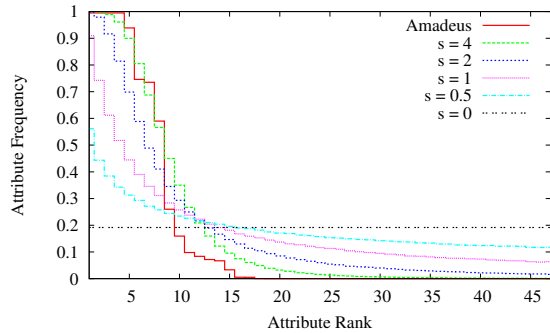


Figure 7: Synthetic Workload Chart
 $N = 47, D = 0, \text{ Vary } s$

8.4 Results Part I: Robustness and Scalability

Unless otherwise indicated, all experiments were run on 15 GB of Ticket data, hash partitioned by flight number; and using the Clock Scan algorithm. All data points were averaged over 5 runs of at least 5 minutes each. By *query latency* and *update latency* we will refer to the time from a query/update being generated by the benchmark driver, to the benchmark driver receiving the full result.

8.4.1 Multi-core Scale-up

In the first experiment, we were interested in the scalability of Crescando on modern multi-core platforms. For this purpose, we saturated the system with a read-only variant of the Amadeus workload. We used *round-robin* partitioning, which means that every scan thread had to process every incoming query. The peak throughputs using Classic, Elevator, and Clock Scan are shown in Figure 8. Classic Scan scales from 0.2 queries/sec for 1 thread to 1.9 queries/sec for 15 threads. In comparison, Elevator Scan scales from 0.8 queries/sec to 10.5 queries/sec. Elevator Scan scales better because more computation is done at each record, making the algorithm less memory-bound than Classic Scan.

Still, the throughput of Elevator Scan is minuscule in comparison with Clock Scan, which scales from 42.3 to 558.5 queries/sec. This is over two orders of magnitude higher than the throughput of Classic Scan. What is more, the linearity of the curve demonstrates that Clock Scan is *CPU bound* on modern NUMA machines. The bump in throughput between 9 and 10 threads is due to the fact that for 10 threads the segments have become small enough to be allocated on the local NUMA node, giving maximum NUMA locality as described in Section 6.

The high performance of Clock Scan compared to Elevator Scan is explained by a large fraction of queries being filtered by a hash index on flight number. Such a scenario is common in on-line query processing, where many queries contain selective predicates on key attributes. Note also that we limited the number of queries to share an Elevator Scan cursor to 32, and to share a Clock Scan cursor to 512. This gave a median query latency of around 1 second for both algorithms, for a fair comparison.

8.4.2 Query Latency: Variable Query Load

In the second experiment, we investigated the robustness of Crescando to bursts in query volume. For this purpose, an increasing load of Amadeus queries was created as we were measuring the 50th, 90th, and 99th percentile of query latency. Figure 9a shows the results. Notice that the through-

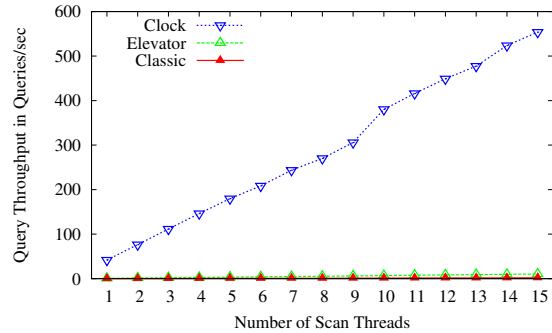


Figure 8: Multi-core Scale-up: Query Throughput
 Amadeus Read-Only, RR Part., Vary Scan Threads

put numbers that Crescando sustains are higher than the peak throughput of the previous experiment, because 99.5% of the Amadeus queries match the hash partitioning.

As for latency, one can see that it is very much constant up to about 200 queries/sec. We found that this is the point where the working set of Clock Scan (indexes and less-selective queries) exceed the L1 cache. Between 200 and about 2,000 queries/sec, latency is logarithmic in the number of queries. Beyond 2,000 queries/sec, latency increases sharply. We ran a number of micro-benchmarks and found that 2,000 queries/sec is the point where the working set exceeds the L2 cache. At about 2500 queries/sec, the system fails to sustain the load and input queues grow faster than queries can be answered.

8.4.3 Query Latency: Variable Update Load

Next, we tested the robustness of Clock Scan to concurrent updates. We pushed a constant 1,000 Amadeus queries per second into the system, and gradually increased the update load to the point where the system could not sustain it (2,300 updates/sec), while we were measuring the 50th, 90th, and 99th percentile of query latency.

As Figure 9b shows, query latency increases by about 35% between 0 and 1,000 updates/sec. Beyond this point, latency hardly grows further. Notice that query latency does not increase sharply right before Crescando fails to sustain the load (2,300 updates/sec). This is because Clock Scan maintains separate input queues for queries and updates.

We conclude that Crescando is a very update-friendly system, especially because the hash partitioning on flight number only helps with the *query load*. Updates do not have a predicate on flight number, but on booking number.

8.4.4 Query Latency: Variable Query Diversity

Crescando is designed to answer any query at any time with predictable latency and impact on the system. This implies a certain robustness to variance in query predicates. To put this property to the test, we measured the sensitivity of throughput and latency to synthetic read-only workloads with varying parameter s (cf. Figure 7).

To keep latency within reason for low values of s , we limited the number of queries to share a scan cursor to 512. The resulting query latencies are shown in Figure 9c. Throughput charts had to be omitted for space reasons. Our measurements include 2,000 queries/sec for $s = 2$, over 300 queries/sec for $s = 1$, and around 50 queries/sec for $s = 0$.

Latency stays in the region required by our use case (cf. Section 2.1) for s up to 1.5. Beyond this point, latency

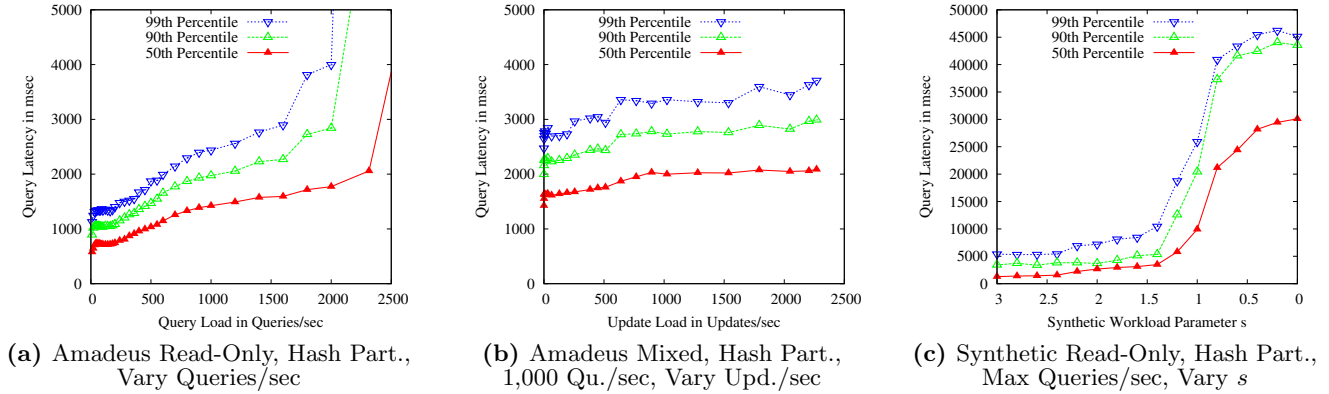


Figure 9: Robustness to Variable Workload: Query Latency

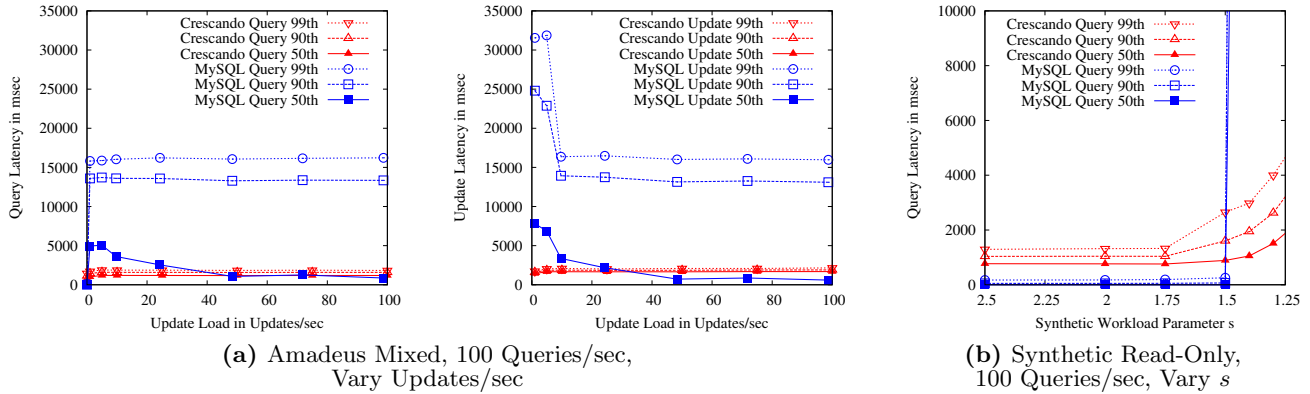


Figure 10: Crescando vs MySQL: Query and Update Latency

increases quickly. To put things into perspective: $s = 0$ represents a uniform distribution of predicate attributes. The workload changes radically between $s = 1.5$ and $s = 0.5$ as Figure 7 indicates, so the latency increase is more the result of a “workload cliff” than a “performance cliff”. The results of the experiment in Section 8.5.1 help to interpret the quality of the results here, as it compares Crescando to traditional solutions under increasing query diversity.

We conclude that performance is reasonable even for unreasonable workloads. Adding a new, minor use case and respective queries to a Crescando installation should *not* violate latency requirements of existing use cases.

8.5 Results Part II: Crescando vs MySQL

To put the performance of Crescando into perspective, we decided to compare it to a popular traditional solution: MySQL 5.1 and its *Memory* table engine. Clearly, this is not an apples-to-apples comparison. More important than absolute numbers is the *shape* of the performance curves. Other existing solutions may reach significantly better performance for specific workloads; e.g., Vertica, MonetDB, or SAP T-Rex for read-mostly workloads. These systems, however, are not applicable to the update-heavy workloads of Amadeus and our evaluation.

Among main-memory databases, we believe the MySQL Memory engine is a good representative for comparison, as most main-memory databases are also based on locking. There are a few commercial main-memory databases which feature optimistic concurrency control (SolidDB, IBM ObjectGrid). However, optimistic concurrency control does not avoid conflicts, particularly between full-table scans and

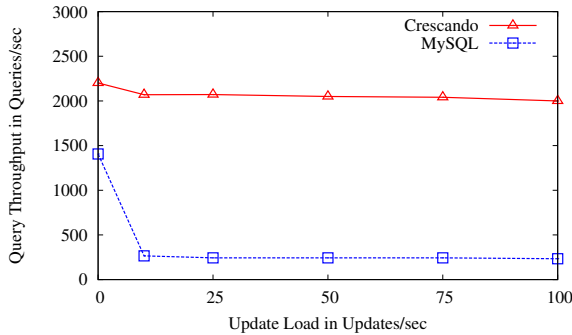
concurrent updates, leaving little reason to believe the results would be significantly better.

Read-write contention could be reduced by multi-version concurrency control (MVCC) schemes, such as snapshot isolation. Unfortunately, no main-memory based database system features MVCC to the best of our knowledge. We experimented with a commercial disk-based database system which implements snapshot isolation. But even when manually tuning this database for our specific workloads, and even when the whole database fit into main memory, the commercial system showed an order of magnitude lower throughput than MySQL for read-only, index-hitting workloads. We found that disk-based databases pay a high price for copying data around in main memory and decided that this precluded a meaningful scientific comparison. Thus, we do not show results for the commercial solution.

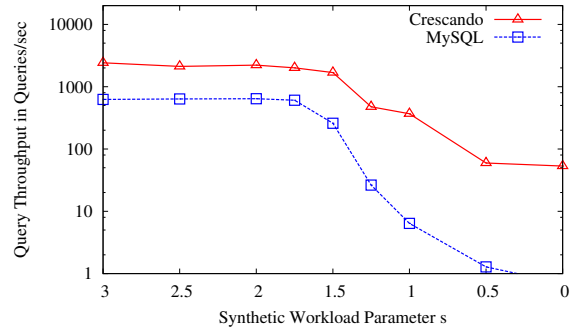
We configured MySQL as follows: an index on <flight number, departure date>, plus an index on booking number (these are the indexes used by Amadeus for their live system); and logging disabled. We spawned 16 client threads, of which 15 were pure readers and 1 was a pure writer. This configuration gave maximum throughput. Additional indexes or threads did not improve performance any further.

8.5.1 Query and Update Latency

In this experiment, we compared the ability of Crescando and MySQL to comply with latency requirements under increasing update load and query diversity. First, we measured query latency for a mixed Amadeus workload with 100 queries/sec and a variable update rate of 0 to 100 updates/sec. The results are shown in Figure 10a.



(a) Amadeus Mixed, Max Qu./sec, Vary Upd./sec



(b) Synthetic Read-Only, Max Queries/sec, Vary s

Figure 11: Crescando vs MySQL: Query Throughput

For *read-only* workloads, query latencies of MySQL are superior (1/13/117 ms for the 50th/90th/99th percentile respectively). With a *single* update, latencies surge to 5,000/13,600/15,800 ms for queries, and 7,800/24,800/31,550 ms for updates. We analyzed the Amadeus workload and found that only one out of 3,500 queries does not hit the indexes. The few full-table scans caused by these queries suffice to cause massive read-write contention and update queuing. MySQL starves writers as long as there are queries coming in, so the benchmark driver periodically has to stop pushing queries into MySQL, which in turn causes arriving queries to queue up. The resulting system behavior is one where queries and updates are processed in batches.

Incidentally, the 16 seconds which are the 99th percentile of query latency and half the 99th percentile of update latency are roughly the time it takes MySQL to do a full-table scan. The decreasing median for increasing update rates is explained by convoy effects.

In comparison, the query latencies of Crescando follow a near-constant curve, similar to Figure 9b. In all cases, query and update latencies are below 2,150 ms.

Next, we injected a synthetic read-only load of a constant 100 queries/sec, and varied the shape parameter s between 2.5 and 1.25. Figure 10b shows the results. Latency for MySQL is superior for $s \geq 1.5$ but surges for $s < 1.5$. Note that at $s = 1.25$, over 98% of the queries still hit the indexes. Again, in comparison, Crescando’s performance degrades gracefully, following a curve similar to Figure 9c.

In summary, query and update latencies of Crescando are significantly more robust to concurrent updates and query diversity than those of MySQL. Performance of Crescando is superior except for read-only workloads where all queries hit the available indexes.

8.5.2 Query Throughput

For the final experiment, we return to the sketch shown in the introduction, Figure 1. To verify the claims made, we first measured the maximum query throughput of both systems for a mixed Amadeus workload with variable update load. Then, we measured the throughput for a synthetic read-only workload of variable s .

As Figure 11 reveals, MySQL’s throughput comes close to Crescando only for a read-only Amadeus workload. For all synthetic workloads, or as soon as updates are added, Crescando outperforms MySQL by one or more orders of magnitude. MySQL’s low query throughput even for high values of s is due to the synthetic workload containing a higher fraction of range queries than the Amadeus work-

load. But more interesting than the absolute numbers are the shapes of the curves. They match those of Figure 1.

Summing all up, we conclude that Crescando can guarantee high throughput and low latency numbers with minimum administration effort, even if the workload is fluctuating or rapidly evolving. If necessary, throughput and latency goals can be met by adding hardware alone, owing to Crescando’s scalability.

9. RELATED WORK

Crescando builds on a number of ideas originating in the data-warehousing, stream-processing, and distributed database domains. These include cooperative scans, scan-only query processing, query-data joins, and shared-nothing architectures. Crescando adapts and extends these ideas to achieve predictable performance for unpredictable workloads in on-line query processing.

Cooperative scans originate in the data-warehousing domain and have been implemented for disk-based database systems such as DB2 UDB [14] and MonetDB/X100 [25], with the goal of sharing disk bandwidth and maximizing buffer-pool utilization across queries. Recently, cooperative scans in *memory* have attracted attention in the data warehousing domain, fueled by the advent of COTS multi-core machines and scans’ inherent parallelizability. Raman et al. [18] and their blink system have demonstrated that main-memory scans can be made a reasonably efficient access path with predictable, low (“constant”) latency.

A follow-up paper on blink by Qiao et al. [17] investigates the optimization problem of sharing a scan cursor between a subset of pending queries with `GROUP BY` clauses. In blink, `GROUP BY`s are implemented through hashing (“agg-tables”), which implies a certain working set associated with each query, turning this into a bin-packing problem.

In contrast to blink, Crescando was designed for on-line query processing, which means large numbers of non-grouping, selective queries over live, concurrently updated data. The resulting multi-query optimization problem is related to but qualitatively different from the one addressed by blink and disk-based systems. The reason is that the cost of scan-based on-line query processing is dominated by predicate evaluation. Optimizing the evaluation of single queries has previously been studied by Ross [20, 21]. In this paper, we extended the idea to sets of queries, and introduced cache-conscious query/update-data join algorithms based on predicate indexing: Index Union Join and Index Union Update Join. The term “query-data join” originates in stream processing. Chandrasekaran and Franklin [6] were the first to

explicitly model sets of query predicates as relations.

In Crescendo, a multi-query optimizer decides which indexes to build using greedy heuristics inspired by publish-subscribe systems such as Le Subscribe [9]. In contrast to data-streaming and publish-subscribe systems, Crescendo not only indexes query predicates, but also update and delete predicates. The necessary selectivity estimates are obtained through probabilistic counting as described by Flajolet and Martin [10], and Whang et al. [23].

Finally, architecting single database instances as a shared-nothing system has been studied using specialized hardware during the era of parallel database systems, most notably in the Gamma Project [7] and Prisma/DB [2]. Since the number of cores and memory controllers, and thereby the penalties for treating memory uniformly, are bound to increase, we expect shared-nothing to become the dominant architecture even for centralized databases. Given the inherent parallelizability of scans and our experimental results, we are convinced that our ideas can be scaled to large numbers of processors and machines by building on the wealth of research on parallel and distributed database systems.

10. CONCLUSIONS AND FUTURE WORK

This paper presented Crescendo, a scalable relational table implementation designed to perform large numbers of queries and updates with guaranteed access latency and data freshness. At the core of Crescendo is a new collaborative scan algorithm called Clock Scan, which models query processing as a join between queries and updates on the one side and the data on the other side.

Crescendo does not always outperform traditional database techniques based on data indexes. The advantages of Crescendo are its scalability and its ability to give query latency and data freshness guarantees for a wide range of workloads. In particular, Crescendo can sustain high query and update rates, even if the types of queries and updates are not known in advance. Crescendo is currently evaluated in a real-world application scenario (airline reservation systems) and we believe there are many other use cases in the areas of operational business intelligence and real-time warehousing.

The ultimate goal is to run Crescendo in large clusters with hundreds of machines in order to scale to arbitrary data and query volume. To this end, we are currently implementing the aggregators and replication groups of the distributed architecture of Section 3.1. In addition, we are continuing to push the limits of individual storage nodes by investigating novel optimizer heuristics, scheduling policies, and cache-aware data structures.

11. ACKNOWLEDGEMENTS

We would like to thank Jeremy Meyer, Peter J. Haas, and the anonymous reviewers for their helpful comments on earlier versions of this paper. 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).

12. REFERENCES

- [1] A. Ailamaki et. al. Weaving relations for cache performance. In *Proc. VLDB '01*, 2001.
- [2] P. M. G. Apers et. al. Prisma/db: A parallel, main memory relational dbms. *IEEE TKDE*, 4(6), 1992.
- [3] H. Berenson et. al. A critique of ansi sql isolation levels. In *Proc. SIGMOD '95*, 1995.
- [4] P. A. Boncz et. al. Database architecture optimized for the new bottleneck: Memory access. In *Proc. VLDB '99*, 1999.
- [5] P. A. Boncz et. al. Monetdb/x100: Hyper-pipelining query execution. In *Proc. CIDR '05*, 2005.
- [6] S. Chandrasekaran and M. J. Franklin. Streaming queries over streaming data. In *Proc. VLDB '02*, 2002.
- [7] D. J. Dewitt et. al. The gamma database machine project. *IEEE TKDE*, 2(1), 1990.
- [8] D. J. DeWitt et. al. An evaluation of non-equi-join algorithms. In *Proc. VLDB '91*, 1991.
- [9] F. Fabret et. al. Filtering algorithms and implementation for very fast publish/subscribe systems. In *Proc. SIGMOD '01*, 2001.
- [10] P. Flajolet and G. N. Martin. Probabilistic counting algorithms for data base applications. *J. Comput. Syst. Sci.*, 31(2):182–209, 1985.
- [11] A. Guttman. R-trees: a dynamic index structure for spatial searching. In *Proc. SIGMOD '84*, 1984.
- [12] S. Harizopoulos et. al. Qpipe: A simultaneously pipelined relational query engine. In *Proc. SIGMOD '05*, 2005.
- [13] A. Kleen. A numa api for linux. Novell Technical Whitepaper, 2005. http://www.novell.com/resourcecenter/ext_item.jsp?itemId=14444.
- [14] C. Lang et. al. Increasing buffer-locality for multiple relational table scans through grouping and throttling. *Proc. ICDE '07*, 2007.
- [15] D. B. Lomet. Key range locking strategies for improved concurrency. In *Proc. VLDB '93*, 1993.
- [16] C. Mohan et. al. Aries: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM TODS*, 17:94–162, 1992.
- [17] L. Qiao et. al. Main-memory scan sharing for multi-core cpus. *Proc. VLDB '08*, 1(1), 2008.
- [18] V. Raman et. al. Constant-time query processing. In *Proc. ICDE '08*, 2008.
- [19] M. Ronström and L. Thalmann. Mysql cluster architecture overview: High availability features of mysql cluster. MySQL Technical Whitepaper, 2004. <http://www.techworld.com/whitepapers/index.cfm?whitepaperid=5663>.
- [20] K. A. Ross. Conjunctive selection conditions in main memory. In *Proc. PODS '02*, 2002.
- [21] K. A. Ross. Selection conditions in main memory. *ACM TODS*, 29(1), 2004.
- [22] T. K. Sellis. Multiple-query optimization. *ACM TODS*, 13(1):23–52, 1988.
- [23] K.-Y. Whang et. al. A linear-time probabilistic counting algorithm for database applications. *ACM TODS*, 15(2):208–229, 1990.
- [24] W. A. Wulf and S. A. McKee. Hitting the memory wall: implications of the obvious. *ACM SIGARCH Comput. Archit. News*, 23(1):20–24, 1995.
- [25] M. Zukowski et. al. Cooperative scans: Dynamic bandwidth sharing in a dbms. In *Proc. VLDB '07*, 2007.