Predictable Performance for Unpredictable Workloads

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

†Systems Group, Dept. of Computer Science, ETH Zurich, Switzerland
*Amadeus IT Group, Sophia-Antipolis, France

Crescando

- Radically different way of on-line query processing
  - Massively parallel: distributed and multi-core
  - Overcomes the “CPU wall” and “memory wall”
- Ideal for operational B.I.
  - Complex query predicates, high update rates
  - Tight access latency and data freshness requirements
- May inspire to
  - rethink database architecture on modern hardware
  - rethink “fairness” and “isolation” of queries

Agenda

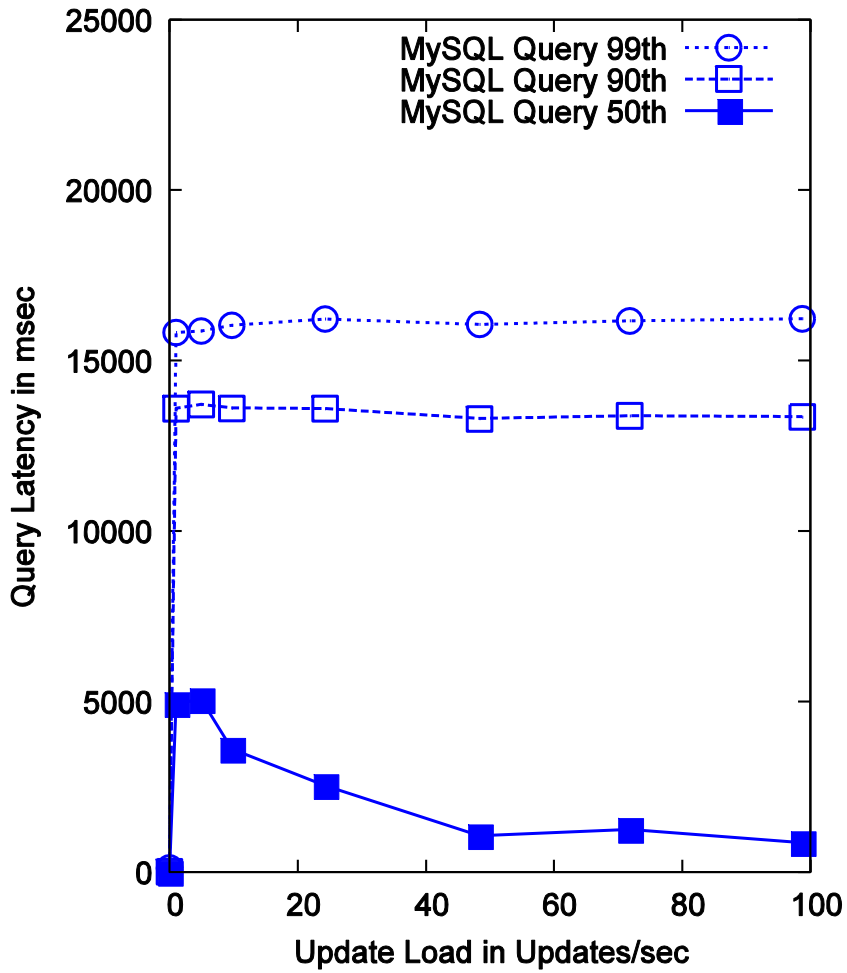
- Background and Use Cases
- Architecture and Framework
- Query Execution
- Performance Evaluation
- Conclusions

BACKGROUND AND USE CASE

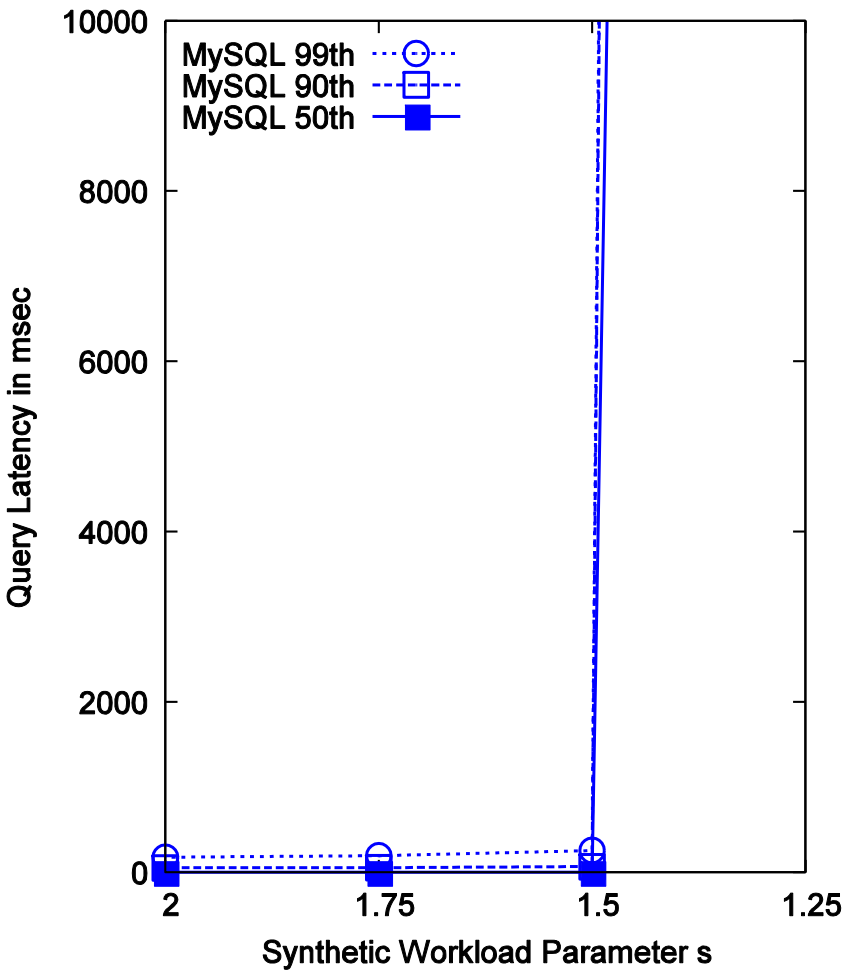
Predictability Matters

- Enterprise workloads evolve in often unpredictable ways
  - New functionality, new services → new queries
  - Ex.: Salesforce allows users to define their own queries
  - Users expect “constant” response time and no performance impact on existing workload
- Hard (impossible?) to achieve in traditional DB systems
  - Designed for best query-at-a-time performance
  - Complex systems with many tuning knobs (indexing, partitioning…)
- Traditional DBs are highly sensitive to
  - Full-table scans
  - Read-write concurrency

Behavior of Traditional DBs



Real-world workload, 100 queries/sec, variable update load



Synthetic read-only workload, variable skew

Crescando

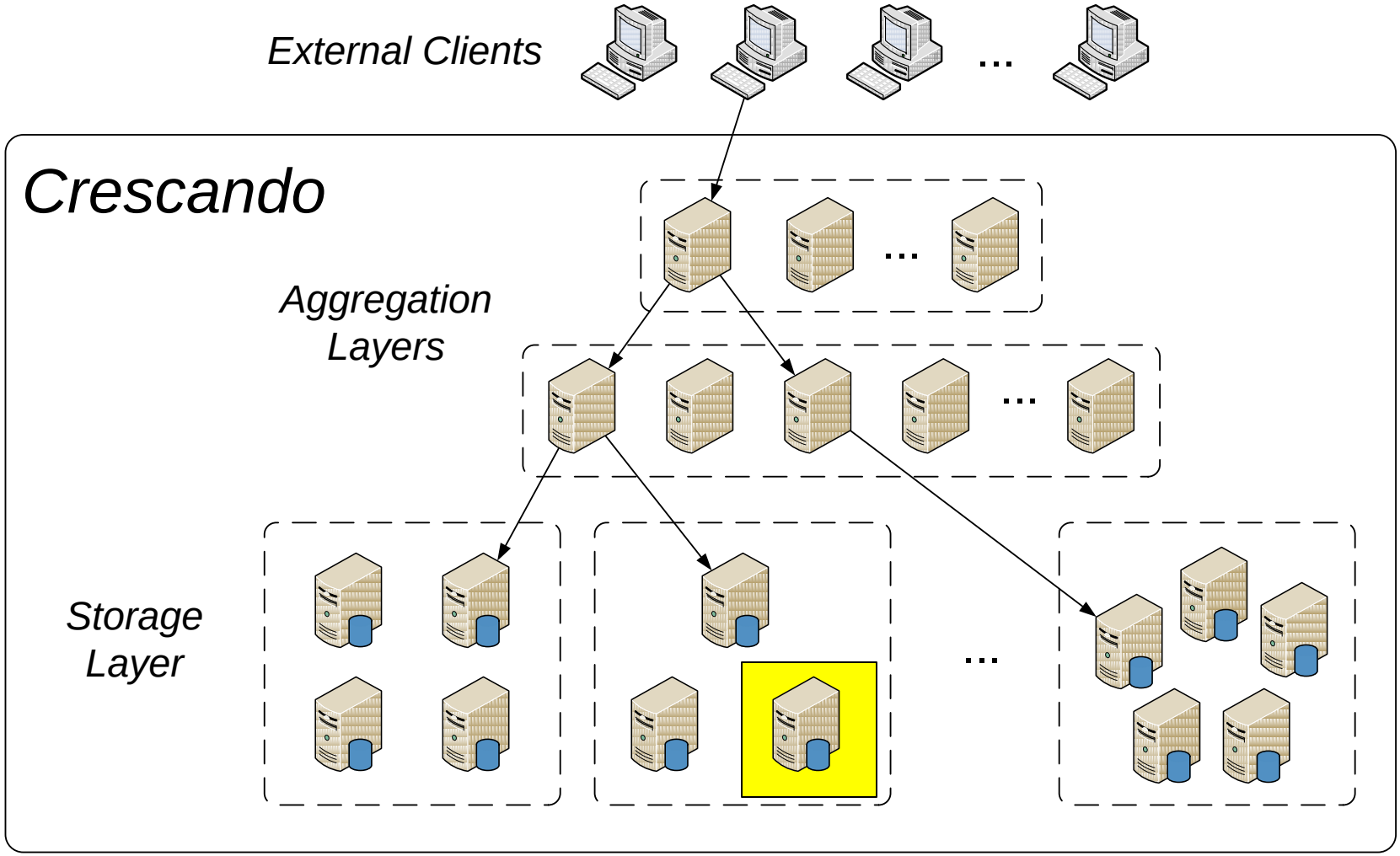
- A **distributed relational table** implementation designed for
  - large numbers of *concurrent* queries and updates with *arbitrary* predicates,
  - with guaranteed access latency and data freshness.

- Amadeus use case
  - "Global Distribution System": large-scale SOA for travel industry
  - Booking table: 100s of updates/sec, unpredictable query predicates, less than 2 second access latency and data freshness
  - Solution: replace *many* existing materialized views on the booking table with a *single* view implemented as a Crescando table

- Many other use cases in operational B.I.
  - Any time the workload is complex, unclear, or changing
  - Any time predictable access latency and data freshness are prime

ARCHITECTURE AND FRAMEWORK

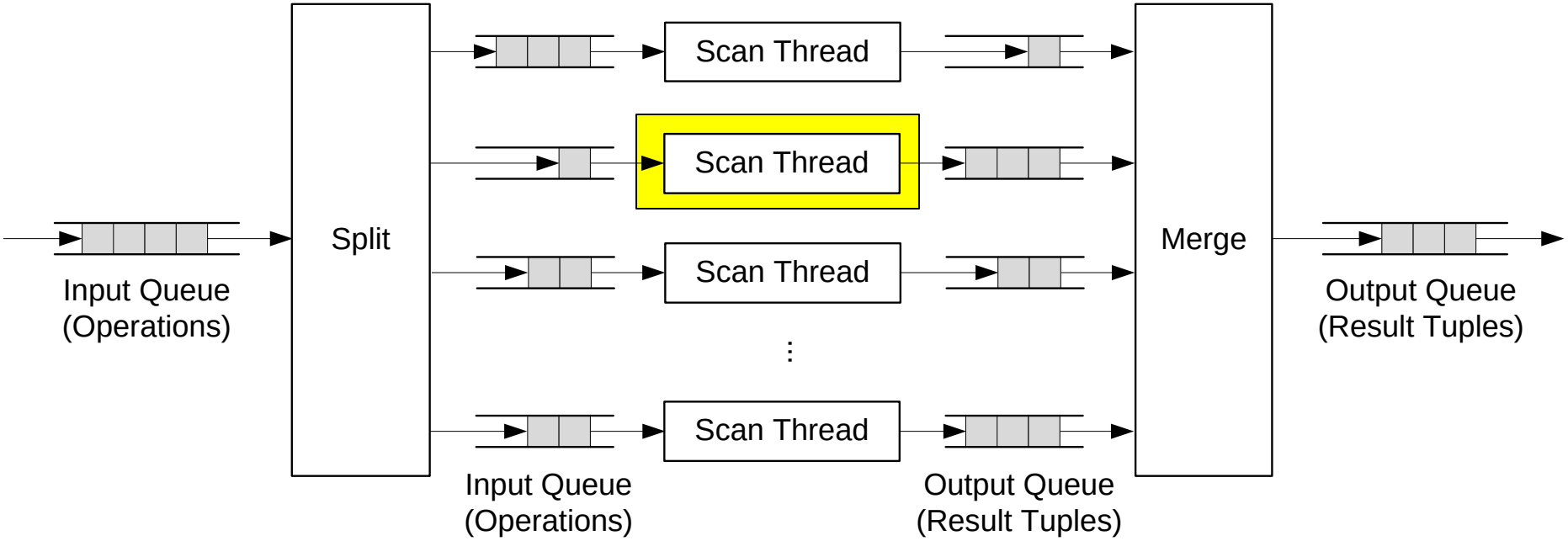
Distributed Architecture



Storage Node Design Points

1. **"Shared nothing"** architecture
   - Horizontally partition the data across cores → multi-core scale-up
2. **Scan-only** query processing
   - No indexes → robust to query diversity
   - Apply updates during scans → no read-write contention
3. **Main-memory** storage
   - High bandwidth (> 10 GB/s per socket on modern machines)
4. **Share** scan cursors
   - Overcomes remaining memory bandwidth bottleneck
5. **Batch** queries and perform **query-data joins**
   - Leads to access latency that is *logarithmic* in query volume (load)
   - Very high peak throughput, stable latency under bursty load

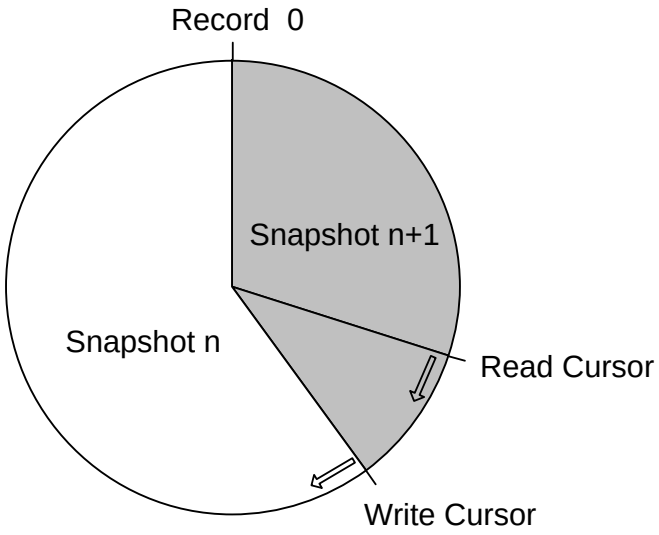
Storage Node Architecture



MULTI-QUERY EXECUTION

Clock Scan

- **Continuously scan** the data partition
  - Logically separate read and write cursors
  - *Activate* pending queries and updates whenever cursors pass record 0



Data Partition

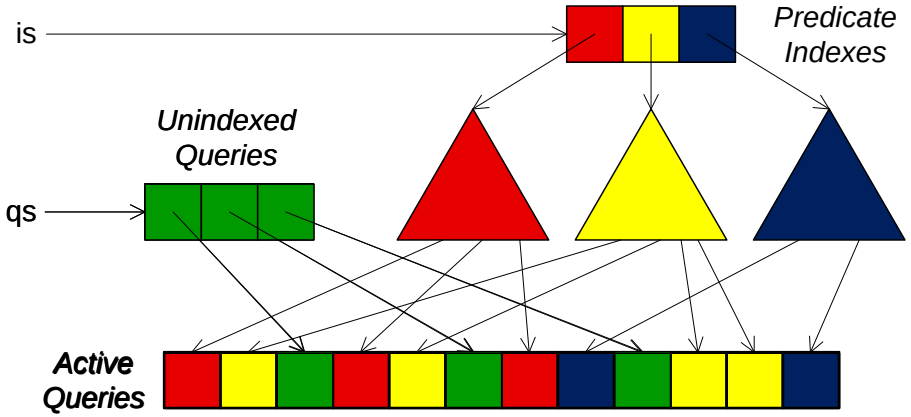
- If
  - queries and updates attach at record 0;
  - cursors cannot pass each other

  then
  - read cursor always sees consistent snapshot;
  - regardless of how query execution is interleaved (→ **query-data joins**).

- Ability to read consistent snapshots of whole partitions enables efficient concurrency control and recovery

Index Union Join

- A **query-data join** algorithm inspired by publish-subscribe systems
  - Joins a *set of records* with a *set of active queries* (1000+)
- Let every query be part of either
  - the set of unindexed queries, or
  - exactly one predicate index



- Algorithm

```
For each scanned record:
    Evaluate each unindexed query;
    Probe each predicate index and evaluate the queries found;
    Union the results (implicit by common output queue).
```

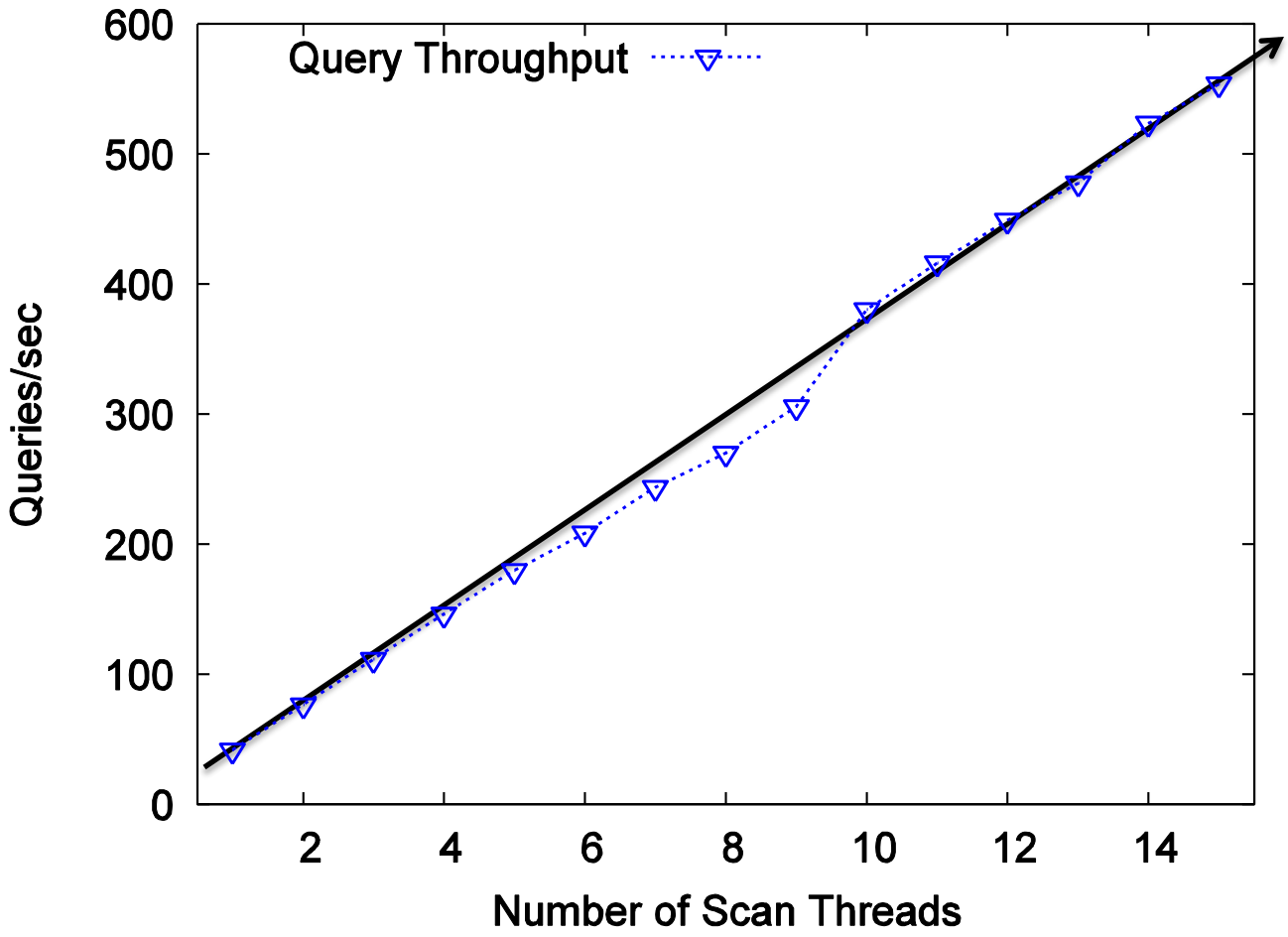
- Can be extended to work for *sequences of relational updates*
  - Index Union Update Join

PERFORMANCE EVALUATION

Setup

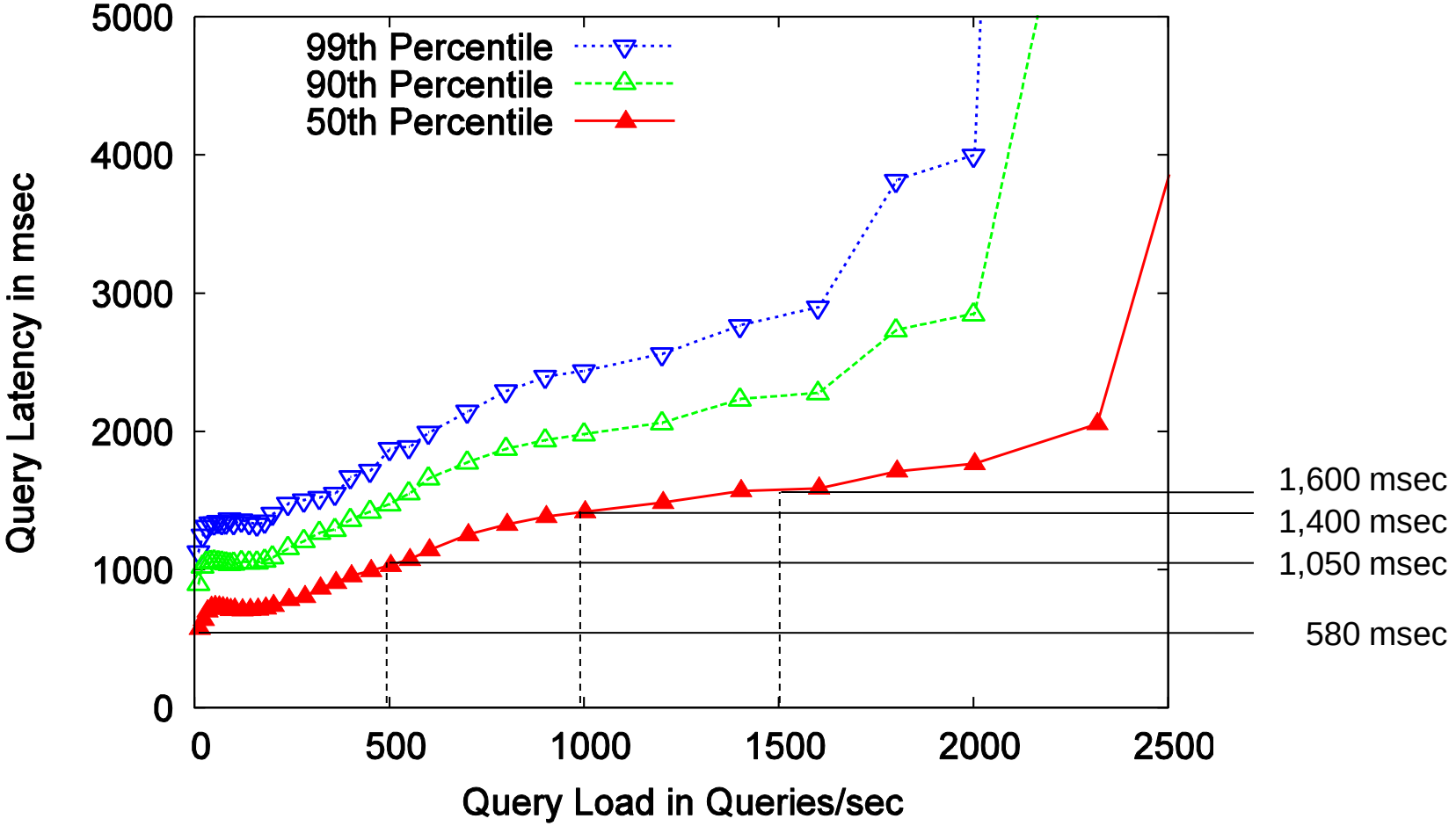
- Storage Node Implementation
  - Shared library for POSIX systems
  - Heavily optimized C++ with some inline assembly
- Platform
  - 16-core AMD Opteron machine with 32 GB DDR2 RAM
  - 64-bit Linux SMP kernel, ver. 2.6.27, NUMA enabled
- Data
  - The Amadeus *Ticket* view (one record per passenger per flight)
  - ~350byte per record; 47 attributes, many of them flags
  - Benchmarks use 15 GB of net data
- 2 workloads
  1. A close simulation of the current, real workload on Ticket
  2. A synthetic workload with variable predicate skew

Multi-core Scale-up



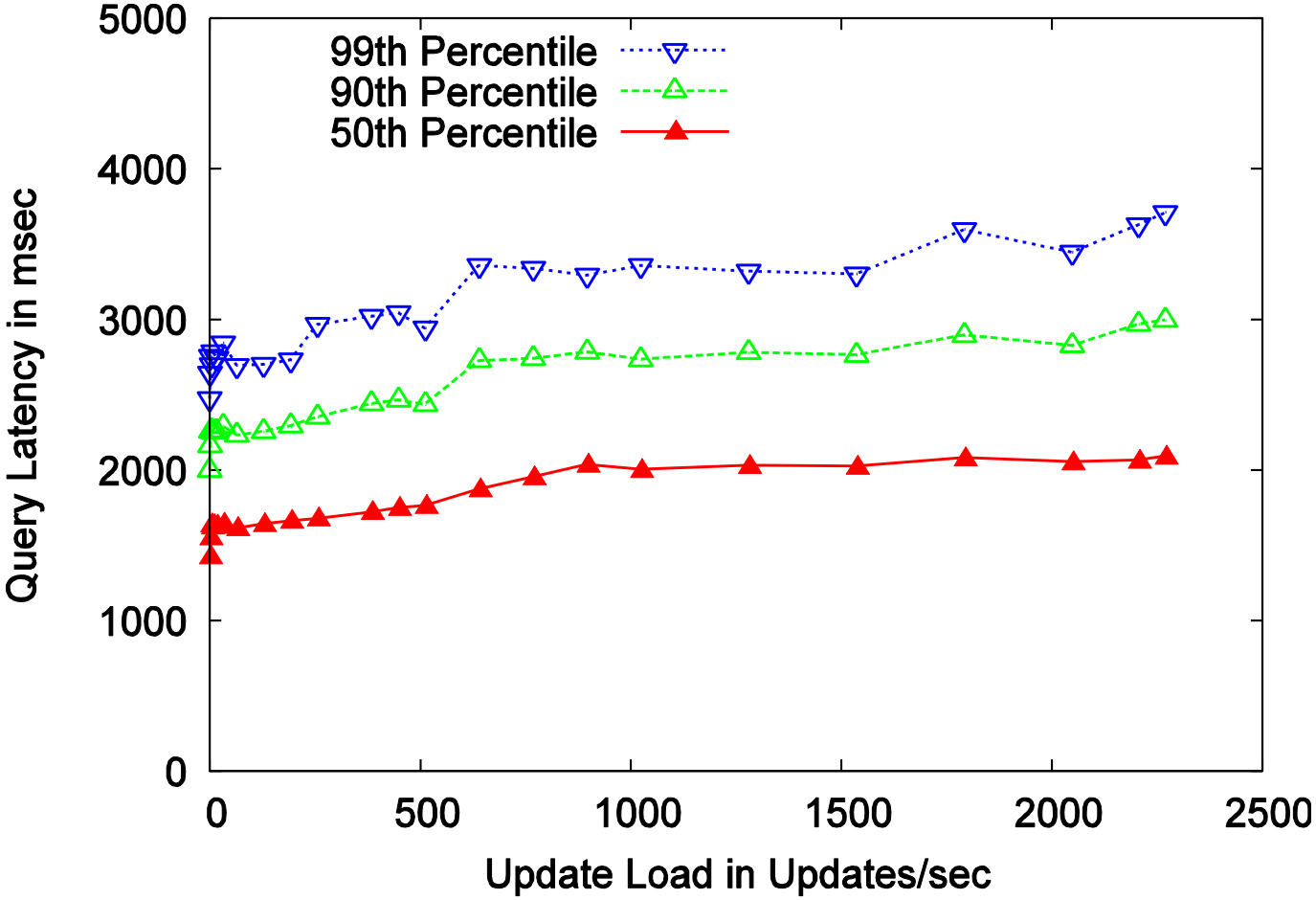
Round-robin partitioning, read-only Amadeus workload, variable number of threads

Latency vs. Query Volume



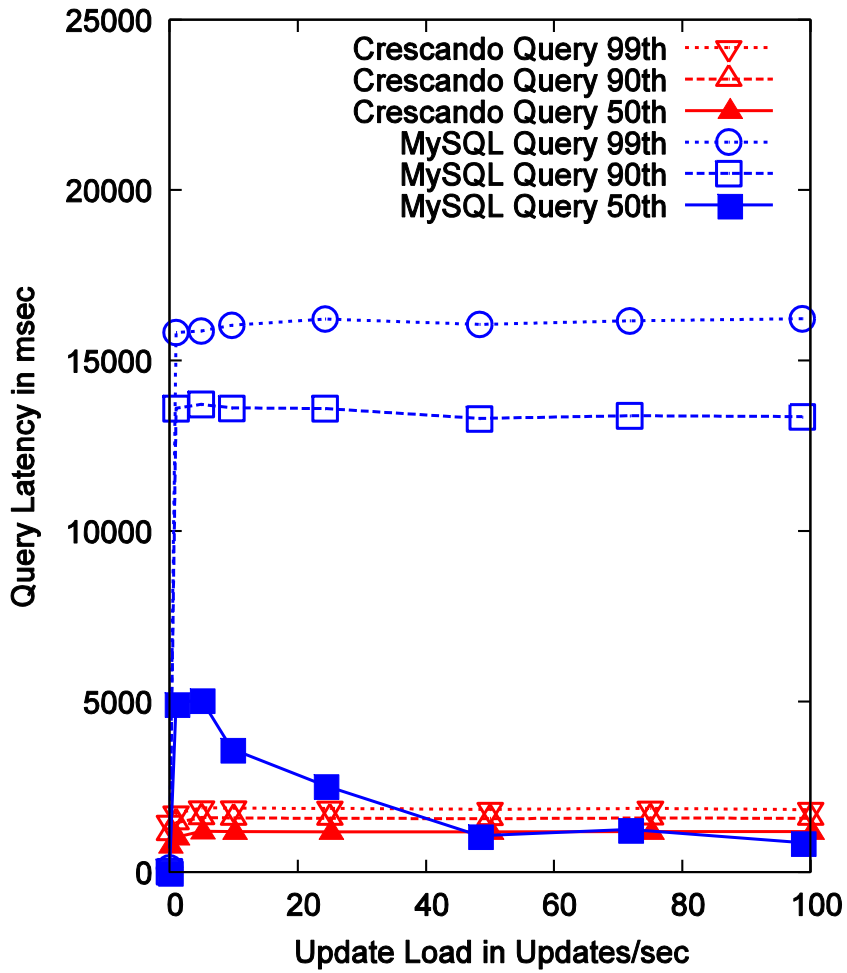
Hash partitioning, read-only Amadeus workload, variable queries/sec

Latency vs. Update Volume

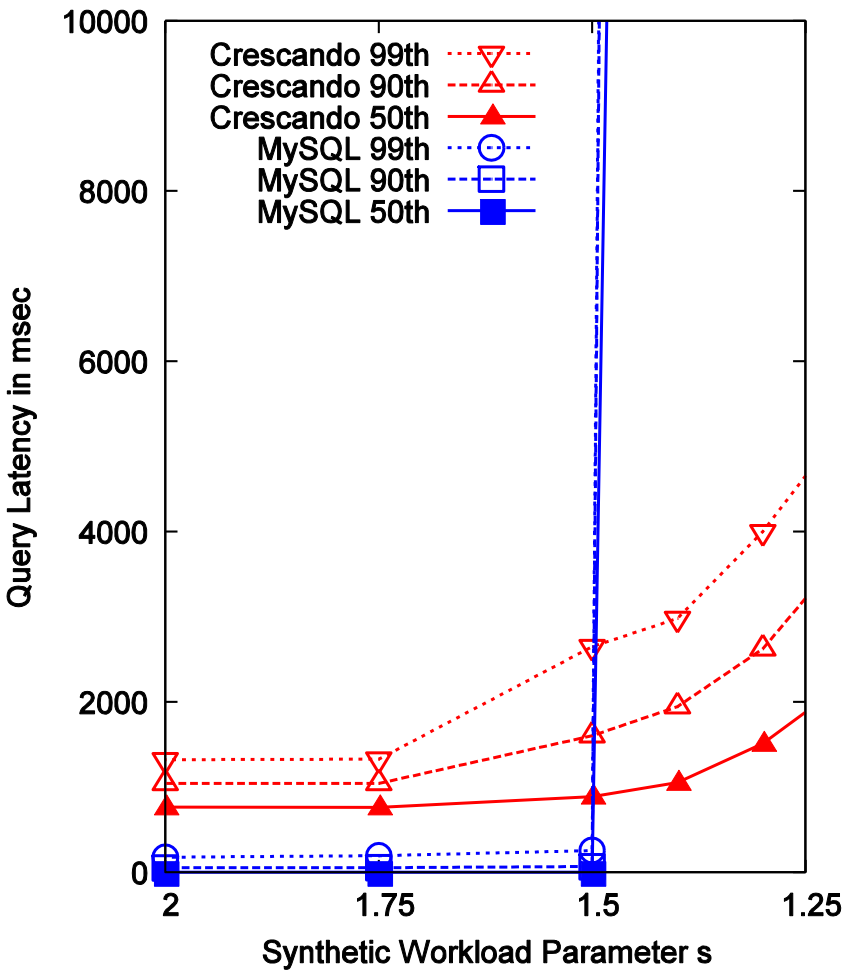


Hash partitioning, Amadeus workload, 1,000 queries/sec, variable updates/sec

Crescando vs. MySQL - Latency

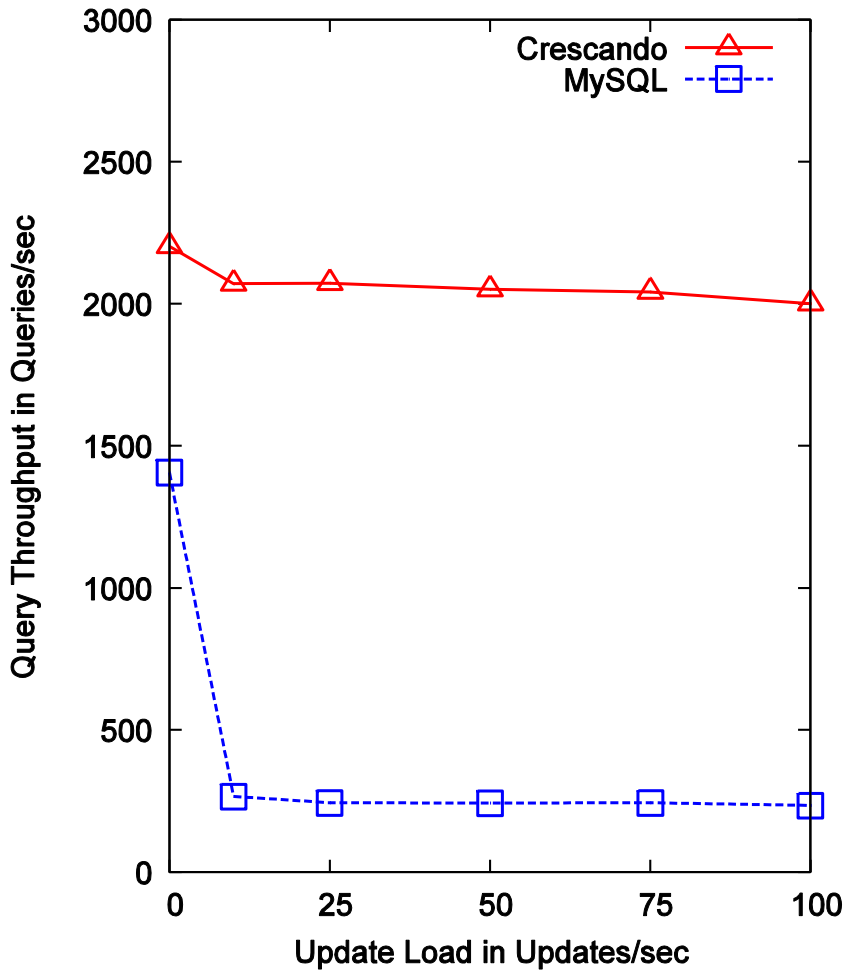


Amadeus workload, 100 queries/sec, variable update load

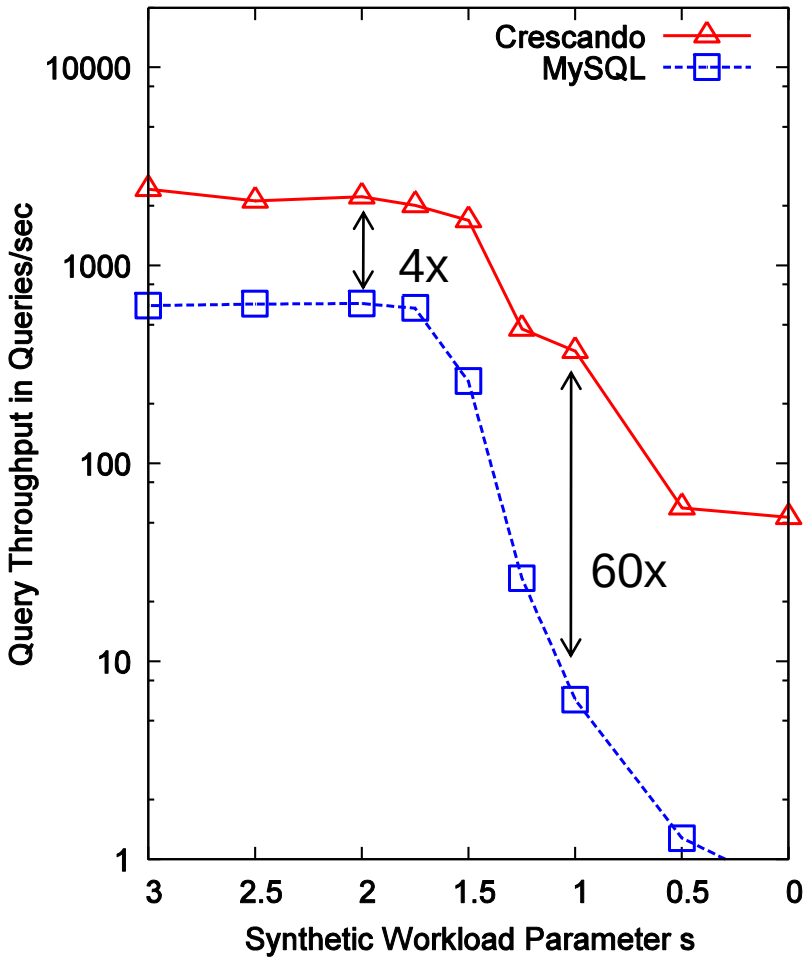


Synthetic read-only workload, variable skew

Crescando vs. MySQL - Throughput



Amadeus workload, variable update load



Synthetic read-only workload, variable skew

CONCLUSIONS

Conclusions

- Crescando: a new way of on-line query processing
  - Massively parallel and scalable
  - Indexing queries rather than data makes optimal use of hardware
- Ideal for operational B.I.
  - Predictable latency and high throughput for mixed and/or scan-heavy workloads
- *Sharing* everything among concurrent queries rather than *isolating* the queries gave predictability
  - Rationale: Law of Large Numbers
  - Rethink traditional notion of “fairness”