SAARLAND UNIVERSITY
COMPUTER SCIENCE
Information Systems Group

Database Systems

WS 08/09

Prof. Dr. Jens Dittrich

Chair of Information Systems Group
http://infosys.cs.uni-saarland.de

Results of Index Evaluation

- yellow: C++
- everything else: Java
- green: original indexes

Team	Time	Technique
Swift	7,02	CSB+ tree
M.Y.Q.M	8,36	bitmap to check whether key exists when looking up; two other improvements,
Illegal Argument	8,64	full CSB+-tree
Team 42	8,95	CSB+ tree
F.M.I.	10,90	CSB+-tree
no_name_3	11,84	B+ tree
we don't like Mondays	11,89	pB+ tree
Original B-Tree	12,08	B+ tree
SCUMM	12,37	tried CSB+-tree but did not succeed, then tried CSS plus some delta but could not make it faster than array
Old-school	12,56	CSB+ tree
Pumuckl	12,74	pB+ tree
t.b.d.	13,98	full CSB+ tree
N.O.R.D.	16,18	CSB+ tree
no_name_1	16,31	Differential index with full CSS tree
Java Tree Map	65,98	red-black-tree
TAB3	n/a	CSB+ tree
D1P2	n/a	CSB+ tree
no_name_2	n/a	CSB+ tree
Team A	n/a	CSB+ tree
Team B	n/a	CSB+ tree
Team C	n/a	CSB+ tree
IEngine	n/a	CSB+ tree

Topics (5/6)

- read-optimized system concepts
  - search engines
  - data warehouses and OLAP
- write-optimized system concepts
  - OLTP
  - publish/subscribe
  - streaming
  - moving objects
- management of geographical data
  - basic concepts
  - GIS, google maps

Other Project-Specific Techniques.

Architectural Outlook

- We have learned a bunch of techniques in the past three months.
- Are DBMSs going to use all of this forever?
- Following discussion based on an invited keynote at VLDB 2007:
  The End of an Architectural Era (It‘s Time for a Complete Rewrite)
  M. Stonebraker, S. Madden, D.Abadi, S.Harizopoulos, N. Hachem, P. Helland
- accompanying paper available on the web

Recap: One Size does not fit all

- today‘s DBMSs can be outperformed by 1-2 orders of magnitude by specialized engines
- this holds **only** for certain applications including
  - data warehousing
    (main-memory parallel column store, e.g. BI Accelerator)
  - stream processing
    (push-oriented operator graphs, e.g., PIPES, Aurora)
  - search on desktop, enterprise, or web
    (inverted lists, e.g., Google, Lucene)

Reasons

- data warehousing
  - query load much different from query load anticipated as “average query“
  - queries read a lot of data (value-oriented processing)
- stream processing
  - push-based processing not well supported by existing DBMSs
  - in theory, triggers could do this job, but: not as efficient
  - stream processing drops the idea that data is stored somewhere
  - everything computed instantly
- search (i.e. IR: Information Retrieval)
  - no transactions, sychronization, recovery
  - simple data structures (inverted lists)
  - SQL not accepted in this community
  - search on top of DBMSs just too slow

OLTP vs. OLAP vs. IR

	OLTP	OLAP	IR
Data Access	read/write	read-mostly	read-mostly
Freshness	up-to-date	stale	stale
Query Access	key-oriented	value-oriented	value-oriented
Indexing	update efficient	**query efficient**	**query efficient**
Data	structured	structured	unstructured
Query optimization	heuristic	cost-based	cost-based
Parallelism	inter-query	intra-query	intra-query
Precision&Recall	1	1	<=1
Data volume	small-medium	huge	huge

DBMSs are the Best Option for OLTP

- Current Assumption:
  - for an OLTP application existing DBMSs are the most efficient engines
- This assumption is **questioned** in the paper.
- The paper argues that **even for an OLTP application** existing DBMS can be beaten by 1-2 orders of magnitude.
- If this is true it means that existing DBMS engines can be beaten in any market!

Reasons

- current RDBMSs were basically invented and designed in the 70ies
- hardware characteristics were much different at that time
  - slow processors (now thousands of times faster)
  - tiny main memories (now thousands of times larger)
  - tiny, slow harddisks (today we may basically keep everything)
  - centralized servers with terminals
- DBMS were designed to work well with disks
- Claim
  - DBMSs should be completely rewritten

OLTP Design Considerations

- Main Memory
- Multi-threading and Resource Control
- Grid Computing
- High Availability
- No Knobs

Main Memory

- today several Gbytes are common
- in a few years a terabyte of main memory not unusual
- today: shared-nothing grid of 20 nodes having each 32 GB (soon 100 GBytes) of main memory costs less than $50,000
- consequence: any database up to one terabyte in size may reside in main memory at reasonable cost
- overwhelming majority of OLTP databases are less than 1 TB
- Consequence: OLTP should be considered a main memory market
  (if not now then within a very small number of years)

Multi-threading and Resource Control

- TPC-C: important OLTP benchmark
- heaviest TA in TPC-C reads about 200 records
- in a main memory system, the net work of a TA consumes less than 1 ms on a low-end machine
- applications, however, must not be allowed to stall TAs
- Therefore: run all commands of a TA to completion single-threaded!
- Consequences
  - code needed to guarantee Isolation **not needed** anymore
  - no multi-threaded (complicated) data structures, no locking
  - a large amount of complex code goes away
- What about “long running“ commands?
  => Not in OLTP if well-designed!
- Note: each CPU core may be considered a separate logical site

Grid Computing

- 1970ies: shared-memory computers
- 1980ies: shared disk
- now: shared nothing
- DBMS should be optimized for this
- no fork-lift upgrade
- For instance
  - if we have N processing nodes
  - now add K new nodes
  - we shouldn‘t need to repartition all data
  - system should do all the magic itself
- Replication and partitioning should be done by the system **while** running TAs
- Correctness of TAs should not be affected.

High Availability: Crashes

- log-based recovery on separate disks/tapes
- tapes (log archive) kept in different geographical site
- today: many companies run hot standbys
  - same machines, same software data in multiple sites
  - if one site crashes: just switch to other machine
  - [Airplanes do it in a similar fashion: two or three independent computer system perform the same computations and vote]
- shared-nothing code was added on top of DBMSs
- however, starting with support for shared-nothing from scratch might be more efficient
- shared nothing converges towards P2P

High Availability: Logging&Recovery

- no redo log
- only undo for local undo of a TA required
- undo log does not have to be persisted beyond the completion of the TA
- therefore undo-log may be a main memory data structure that is discarded on TA commit
- recovery will be accomplished via network recovery from a remote site
  - when crashed machine is up again: get data from a machine that is still running
- Consequences:
  - no need for ARIES-style recovery algorithms
  - again: a large amount of complex code goes away

No Knobs

- DBMS products have many tuning knobs
- manpower (DBAs) needed to find the optimal position of knobs
- some self-tuning efforts
- however, current systems far away from being self-tuning: a skilled DBA will in most cases produces a more efficient knob setting
- How can we come up with a DBMS that does not have any (or just few) visible knobs?

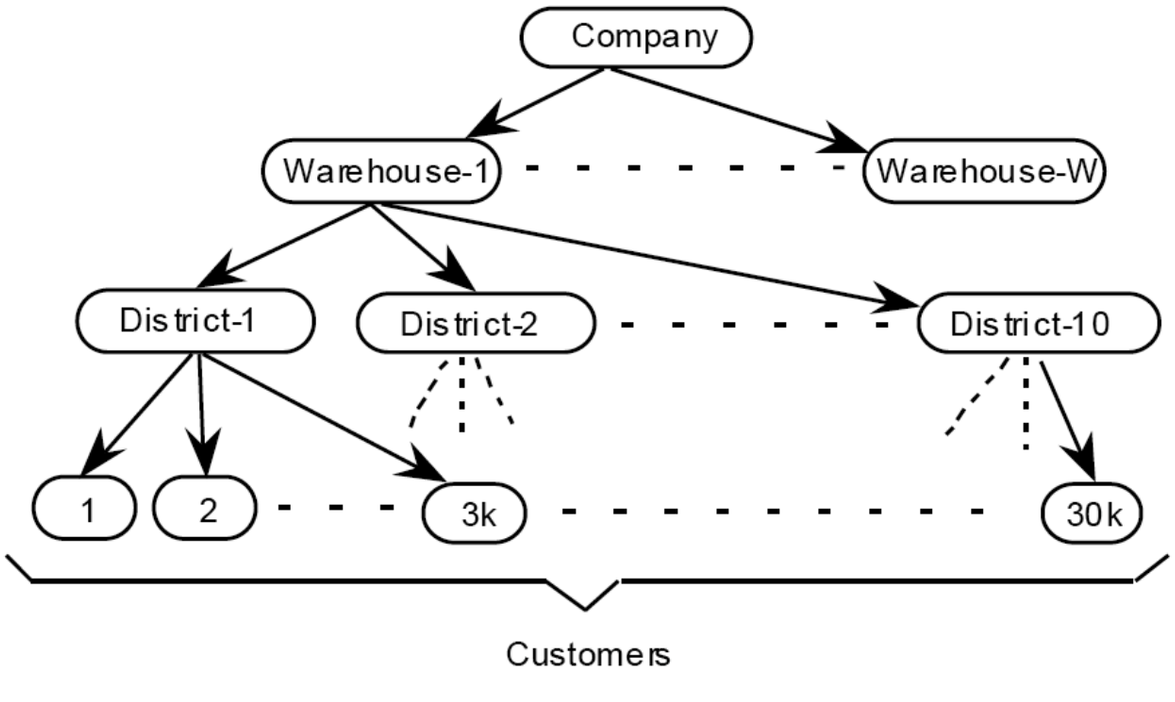
Transaction Processing

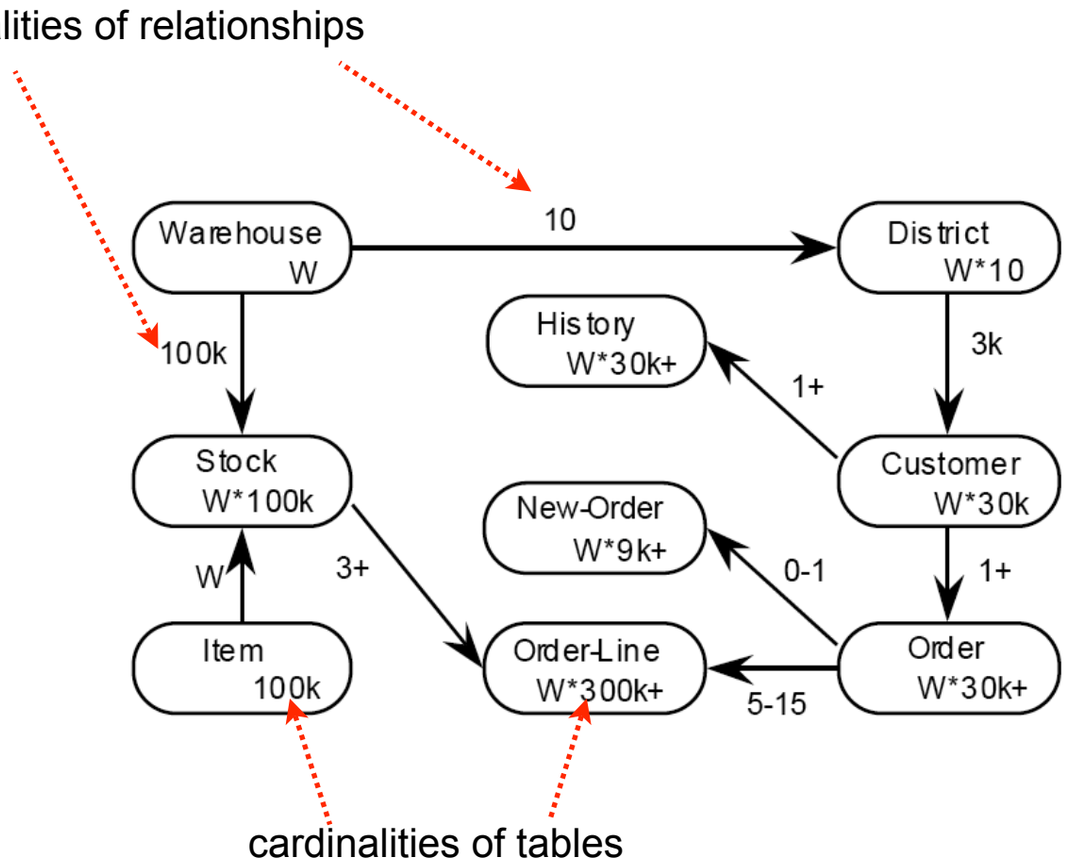
- so far: assume a main memory-based grid system, built-in high availability, no user stalls and useful TA work under 1 ms
- some bottlenecks remain
- still: JDBC/ODBC style client-DBMS interaction is very expensive
  - DBMS code should be executed inside the DBMS avoiding this bottleneck
  - use stored procedures
- two-phase commit protocol for distributed TAs should be avoided wherever possible
  - network latencies for round trip communications too high (milliseconds)

Idea: Analyze Workload

- Idea: analyze schema and query workload beforehand
- Central Observation:
  - in many schemas all tables have exactly one join term which is a 1-n relationship to its ancestor
  - except a single table called **root** table
  - hence, schema is a tree of 1-n relationships
  - if schema is not a tree already it may be transformed into one in many cases
  - Example, TPC-C

TPC-C

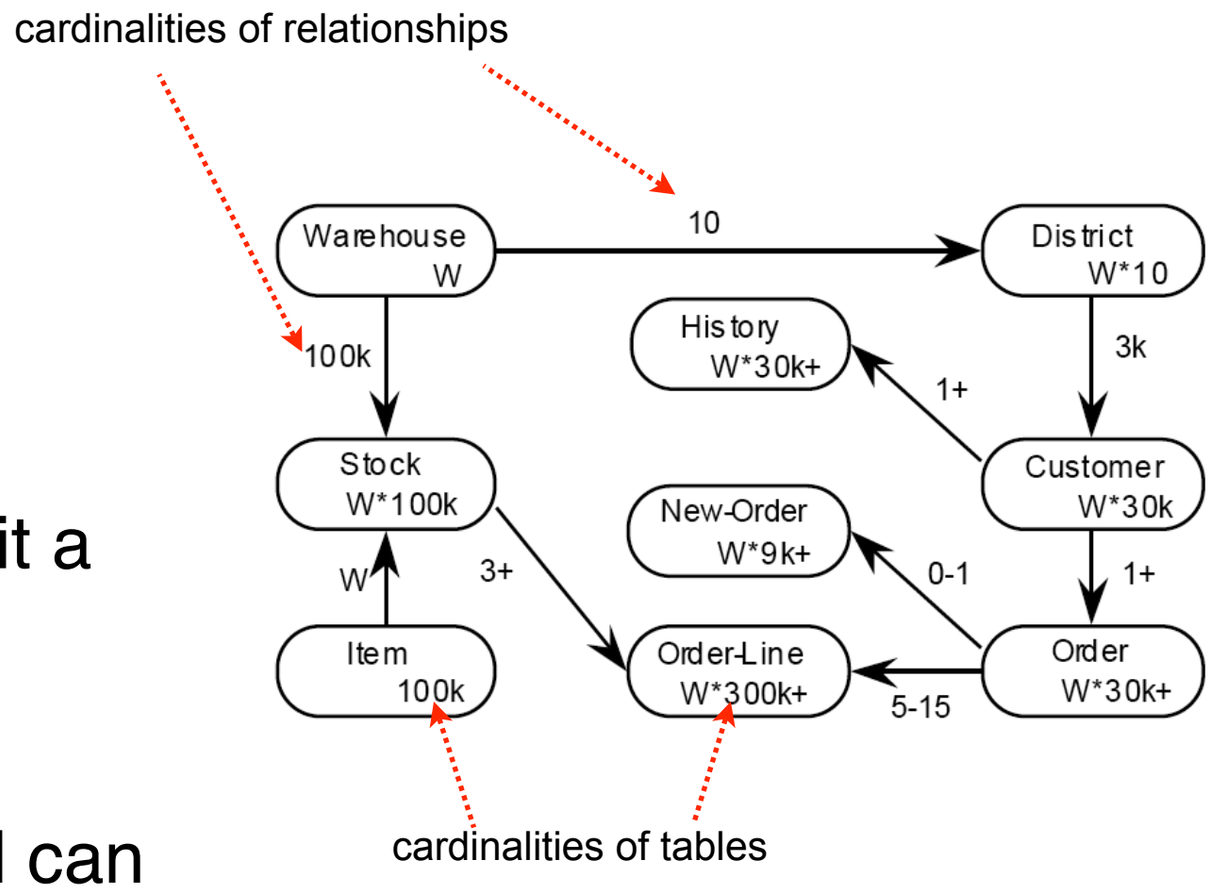




- arrow denotes 1:n relationship
- i.e., a warehouse has 10 districts
- a district has 3K customers, etc.
- W is the number of warehouses (scaling factor)
- if this were a tree-schema, we could easily partition the root (Warehouse) horizontally
- descending tables may be partitioned on foreign keys such that all equi-joins in the tree become local to a site (compare Parallel DB slides)

TPC-C

- however: in TPC-C
  - Item table
  - and relationship Order-Line to Stock make it a non-tree schema



- Solution
  - Item-table is read-only and can be replicated to each site (constant size of 100K items does not hurt much)
  - Orde-Line may also be partitioned according to the Warehouse W (compare Parallel DB slides)
- Therefore combination of partitioning and replication makes this a tree-schema.

Advantages

- each TA is now local to each site
  (however, replicas may have to be synchronized)
- no communication effort among TAs
- general transformation: read-only tables may lead to replicated tables
  (no synchronization cost for updating)

Identify Application and Transaction Classes

- one-shot application:
  - apps that have TAs that do not interfere, i.e., result of one operation does not depend on another operation
  - each site waits a small amount of time to guarantee timestamp order in case multiple machines send commands
  - no redo log, no concurrency control, no distributed commit
- two-phase:
  - first phase: do some read-only operations (based on these operations the TA may be aborted)
  - second phase: do some queries and updates that are **guaranteed** to not violate any integrity constraints (TA may not be aborted in this phase)
  - no undo-log required

Different Degrees of Concurrency Control without Violating Isolation

- core idea: identify TA classes that do not interfere anyway
- obviate concurrency control for these classes
- increase levels of concurrency control for TA classes gradually
- instead of running full-blown concurrency control for each TA
- for (hopefully few) remaining TAs:
  - overhead of pessimistic concurrency control (locking) is expected to be too heavy
  - therefore: run optimistic concurrency control scheme

Initial Experimental Results

- paper reports following results
- TPC-C
- prototype versus commercial DBMS
- both on dual-core 2.8 GHz machine with 4GB main memory and 250 GB disk

- prototype 70,416 TPC-C TAs/sec
- commercial DBMS 850 TPC-C TAs/sec
- factor 82

“SQL is not the Answer“

- authors are pessimistic about SQL
- SQL too complex
- OLTP needs a different set of language features than OLAP
- same applies for streaming data and text (no SQL)
- therefore authors argue that various engines (OLTP, OLAP, text, streams) should provide vertical-market specific languages
- these languages may/should each be simpler than SQL

Two Approaches

- two approaches to integrate programming languages and DBMS
- first: embed sublanguage
  - embed sublanguage (e.g. SQL) into programming language (e.g. Java)
  - tons of problems: i.e., type mismatch declarative versus procedural
  - no optimization across Java and SQL
  - ugly
- second: extend programming language
  - take existing language (e.g., Java, Pyhton, Ruby) and extend it to support data manipulation and access
  - no need to have a standard for sublanguage (i.e., SQL)
  - languages easier to change/extend than SQL
  - use language as stored-procedure language
    (tears down boundary between inside DBMS and outside PL world)

SAARLAND UNIVERSITY
COMPUTER SCIENCE
Information Systems Group

OLTP Through the Looking Glass

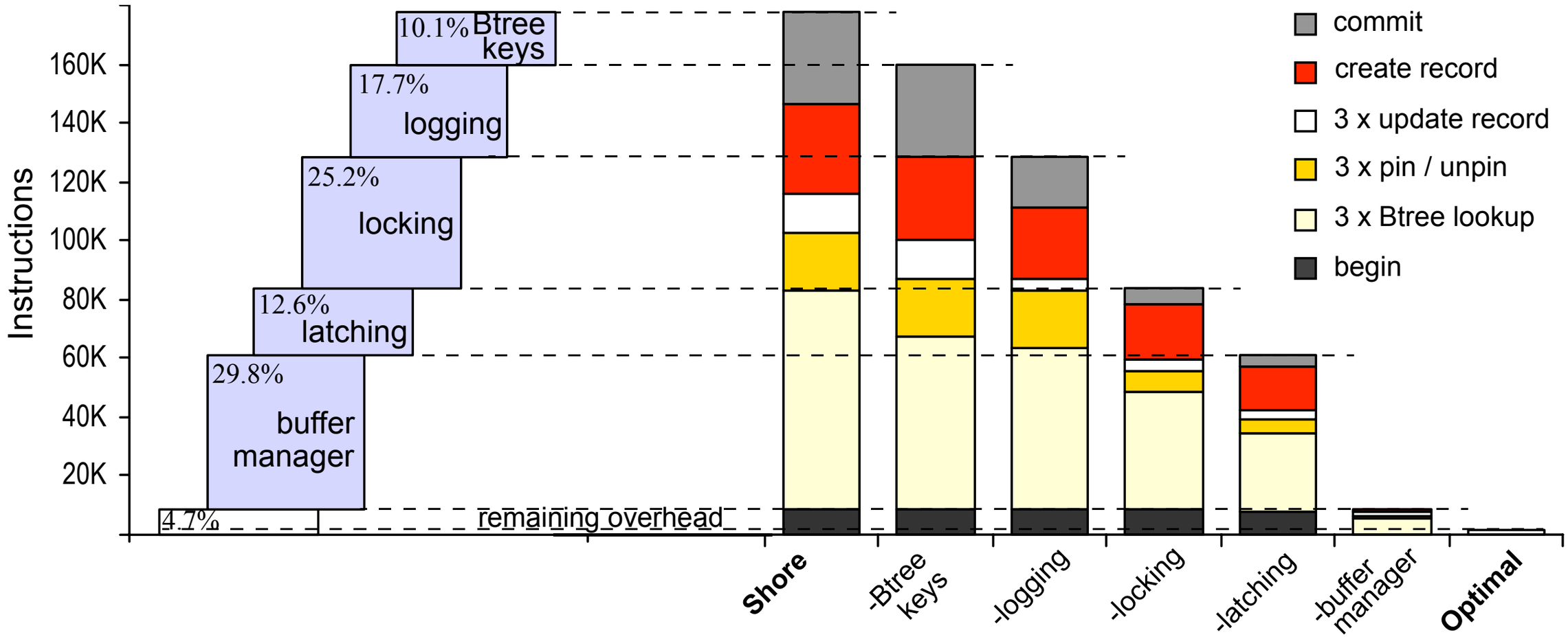
Experimental Study on OLTP Performance

- Stavros Harizopoulos, Daniel J. Abadi, Samuel Madden, Michael Stonebraker: OLTP through the looking glass, and what we found there. SIGMOD Conference 2008
- Idea:
  - take existing disk-based OLTP system (Shore)
  - run the system in main memory, no I/O whatsoever
  - remove “unnecessary“ functionality
  - report how much can be gained

Setup

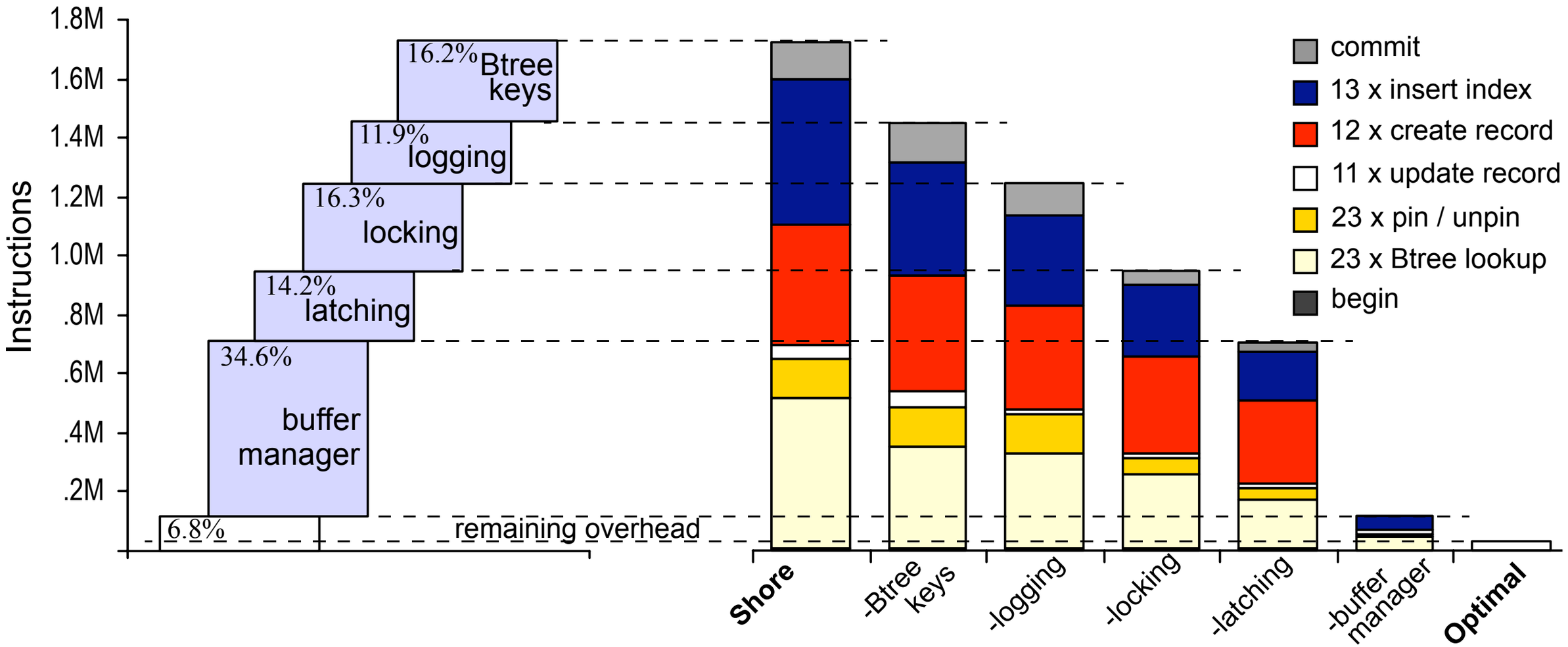
- single-core Pentium 4, hyperthreading diabled, 1GB RAM
- TPC-C benchmark
- instruction and cycle counts using PAPI library
- http://icl.cs.utk.edu/papi/
- used Shore
- placed 11 switches in Shore to disable functionality

TPC-C Payment Transaction



- 20x performance gain when removing everything
- still 6x difference to “**optimal**“
- **optimal**: implementation based on main-memory B+-tree package...

TPC-C New Order Transaction



- main difference:
  - adds B-tree insertions in the mix of operations
  - more relative benefit for Btree key handling improvment
- overall effect similar

Implications for Future OLTP Engines

- removing single components do not provide much benefit
- removing multiple components, however, may provide a factor 20 or more performance gains
- research in various areas required
  - what is the best concurrency control scheme for main memory?
  - how to handle multi-cores? as separate nodes?
  - weaker forms of cosistency, e.g., eventual consistency
  - cache-conscious index structures

Summary

- current DBMS architectures contain several techniques that may be obviated in the future including
  - log-based recovery
  - multi-threading to hide latency
  - log-based concurrency control
  - disk-optimized techniques
- reasons:
  - hardware has changed dramatically
  - there is no “one size fits all“ engine
- impact:
  - great times for researching/inventing new techniques
  - great times for interesting Master‘s Theses...

Next Topic: Yahoo! PNUTS.