Parallel Database Primer

THE · UNIVERSITY · OF · CALIFORNIA · CAL ·

Joe Hellerstein

Today

- **Background:**
  - The Relational Model and you
  - Meet a relational DBMS
- **Parallel Query Processing: sort and hash-join**
  - We will assume a “shared-nothing” architecture
  - Supposedly hardest to program, but actually quite clean
- **Data Layout**
- **Parallel Query Optimization**
- **Case Study: Teradata**

A Little History

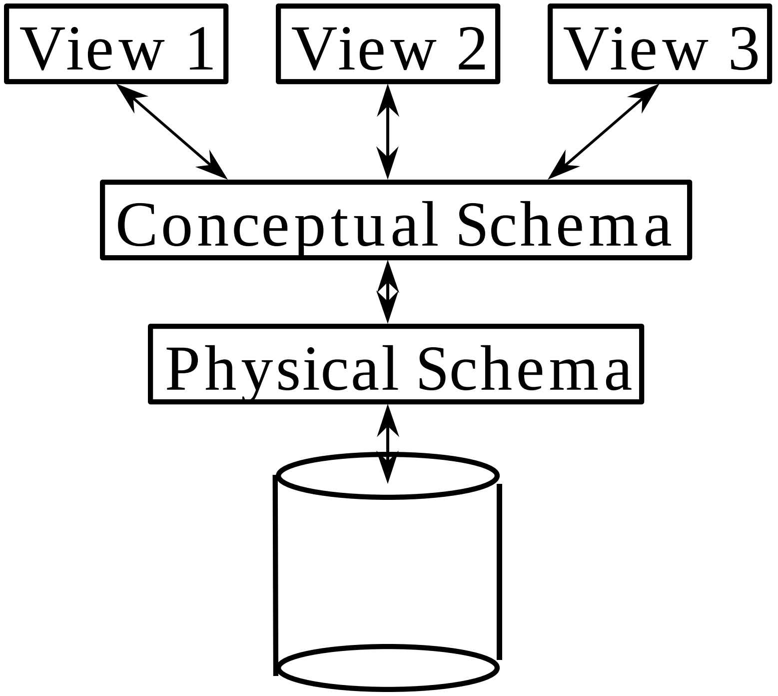
- **In the Dark Ages of databases, programmers reigned**
  - data models had explicit pointers (C on disk)
  - brittle Cobol code to chase pointers
- **Relational revolution: raising the abstraction**
  - Christos: “as clear a paradigm shift as we can hope to find in computer science”
  - *declarative languages* and *data independence*
  - key to the most successful parallel systems
- **Rough Timeline**
  - Codd’s papers: early 70’s
  - System R & Ingres: mid-late 70’s
  - Oracle, IBM DB2, Ingres Corp: early 80’s
  - rise of parallel DBs: late 80’s to today

Relational Data Model

- **A *data model* is a collection of concepts for describing data.**
- **A *schema* is a description of a particular collection of data, using the a given data model.**
- **The *relational model of data* :**
  - Main construct: *relation*, basically a table with rows and columns.
  - Every relation has a *schema*, which describes the columns, or fields.
  - Note: no pointers, no nested structures, no ordering, no irregular collections

Two Levels of Indirection

- **Many *views*, single *conceptual (logical) schema* and *physical schema*.**
  - Views describe how users see the data.
  - Conceptual schema defines logical structure
  - Physical schema describes the files and indexes used.



Example: University Database

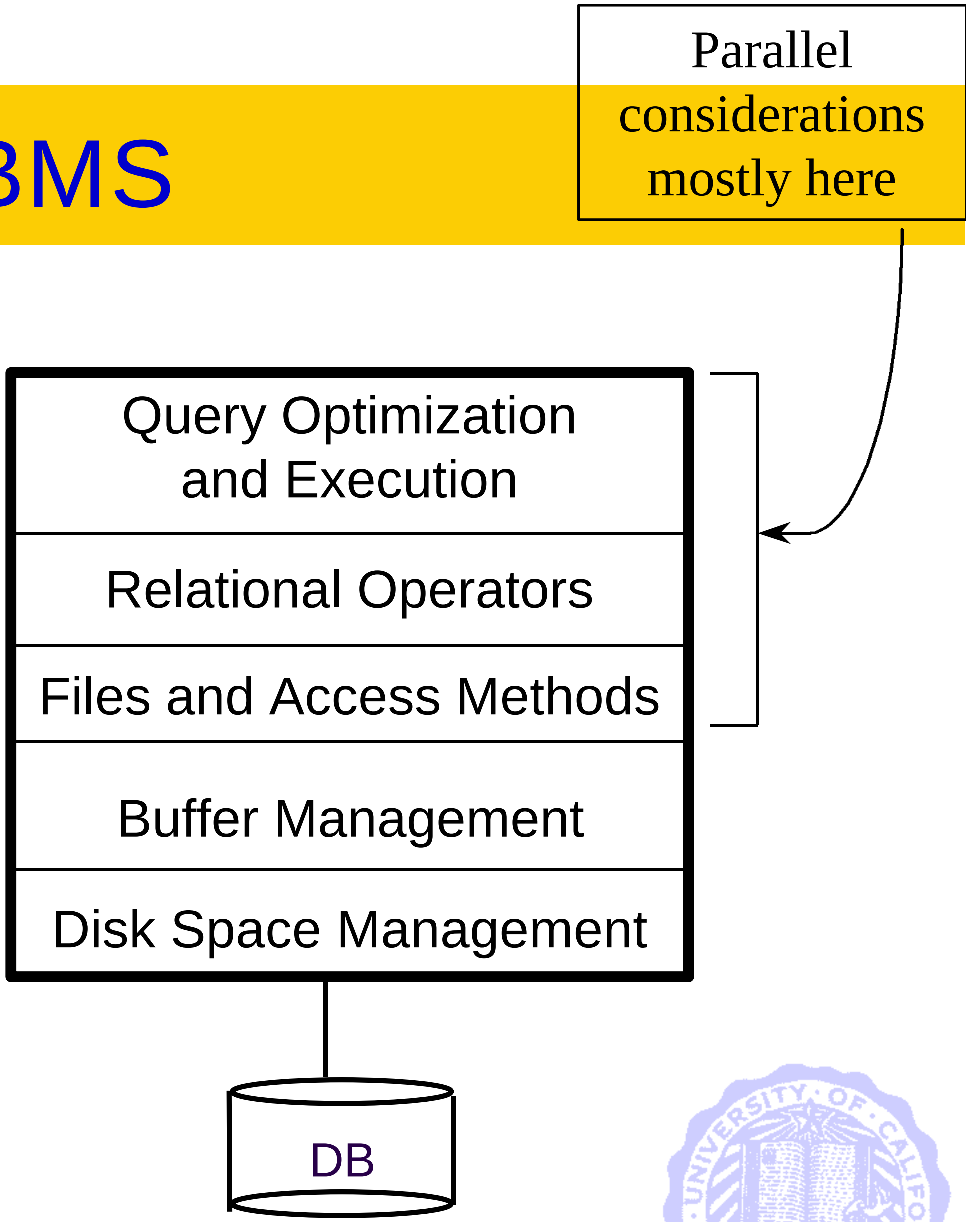
- **Conceptual schema:**
  - *Students(sid: string, name: string, login: string, age: integer, gpa:real)*
  - *Courses(cid: string, cname:string, credits:integer)*
  - *Enrolled(sid:string, cid:string, grade:string)*
- **Physical schema:**
  - Relations stored as unordered files.
  - Index on first column of Students.
- **External Schema (View):**
  - *Course_info(cid:string,enrollment:integer)*

Data Independence

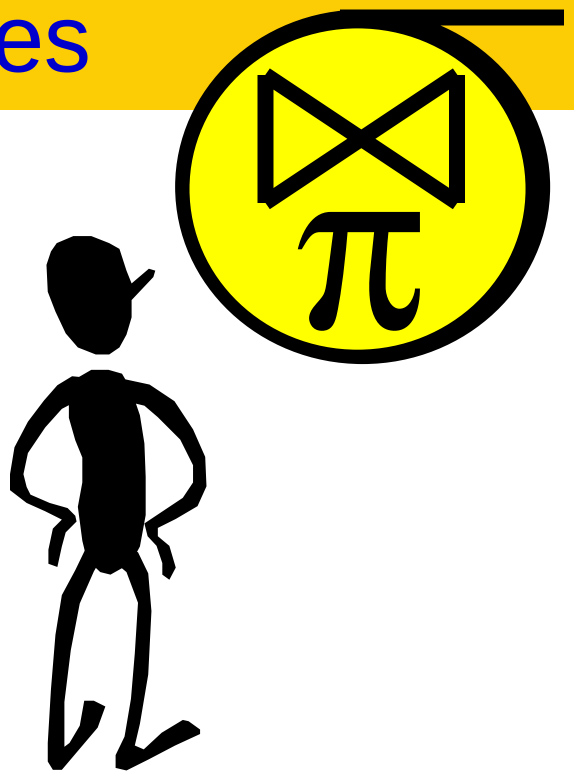
- **Applications insulated from how data is structured and stored.**
- ***Logical data independence*:**
  - Protection from changes in *logical* structure of data.
  - Lets you slide || systems under traditional apps
- ***Physical data independence*:**
  - Protection from changes in *physical* structure of data.
  - Minimizes constraints on processing, enabling clean parallelism

Structure of a DBMS

- **A typical DBMS has a layered architecture.**
- **The figure does not show the concurrency control and recovery components.**
- **This is one of several possible architectures; each system has its own variations.**



Relational Query Languages



By relieving the brain of all unnecessary work, a good notation sets it free to concentrate on more advanced problems, and, in effect, increases the mental power of the race.

-- Alfred North Whitehead (1861 - 1947)

Relational Query Languages

- ***Query languages:*** **Allow manipulation and retrieval of data from a database.**
- **Relational model supports simple, powerful QLs:**
  - Strong formal foundation based on logic.
  - Allows for much optimization/parallelization
- **Query Languages != programming languages!**
  - QLs not expected to be “Turing complete”.
  - QLs not intended to be used for complex calculations.
  - QLs support easy, efficient access to large data sets.

Formal Relational Query Languages

Two mathematical Query Languages form the basis for "real" languages (e.g. SQL), and for implementation:

❶ ***Relational Algebra*: More operational, very useful for representing internal execution plans.**

"Database byte-code". Parallelizing these is most of the game.

❷ ***Relational Calculus*: Lets users describe what they want, rather than how to compute it. (Non-operational, *declarative* -- SQL comes from here.)**

Preliminaries

- **A query is applied to *relation instances*, and the result of a query is also a relation instance.**
  - *Schemas* of input relations for a query are fixed (but query will run regardless of instance!)
  - The schema for the *result* of a given query is also fixed! Determined by definition of query language constructs.
  - Languages are *closed* (can compose queries)

Relational Algebra

- **Basic operations:**
  - Selection (σ) Selects a subset of rows from relation.
  - Projection (π) Hides columns from relation.
  - Cross-product (x) Concatenate tuples from 2 relations.
  - Set-difference (—) Tuples in reln. 1, but not in reln. 2.
  - Union (∪) Tuples in reln. 1 and in reln. 2.
- **Additional operations:**
  - Intersection, join, division, renaming: Not essential, but (very!) useful.

Projection

- **Deletes attributes that are not in *projection list*.**
- ***Schema* of result:**
  - exactly the fields in the projection list, with the same names that they had in the (only) input relation.
- **Projection operator has to eliminate *duplicates*! (Why??)**
  - Note: real systems typically don't do duplicate elimination unless the user explicitly asks for it. (Why not?)

sname	rating
yuppy	9
lubber	8
guppy	5
rusty	10

$$p_{sname,rating}(S2)$$

age
35.0
55.5

$$p_{age}(S2)$$

Selection

- **Selects rows that satisfy *selection condition*.**
- **No duplicates in result!**
- ***Schema* of result:**
  - identical to schema of (only) input relation.
- ***Result* relation can be the *input* for another relational algebra operation! (*Operator composition.*)**

sid	sname	rating	age
28	yuppy	9	35.0
58	rusty	10	35.0

$$s_{rating>8}(S2)$$

sname	rating
yuppy	9
rusty	10

$$p_{sname,rating}(s_{rating>8}(S2))$$

Cross-Product

- **S1 x R1: All pairs of rows from S1,R1.**
- **Result schema: one field per field of S1 and R1, with field names `inherited' if possible.**
  - Conflict: Both S1 and R1 have a field called sid.

(sid)	sname	rating	age	(sid)	bid	day
22	dustin	7	45.0	22	101	10/10/96
22	dustin	7	45.0	58	103	11/12/96
31	lubber	8	55.5	22	101	10/10/96
31	lubber	8	55.5	58	103	11/12/96
58	rusty	10	35.0	22	101	10/10/96
58	rusty	10	35.0	58	103	11/12/96

☛ *Renaming operator*: $\rho\ (C(1 \rightarrow sid1, 5 \rightarrow sid2),\ S1 \times R1)$

Joins

- ***Condition Join***: $R \bowtie_c S = s_c(R \times S)$

(sid)	sname	rating	age	(sid)	bid	day
22	dustin	7	45.0	58	103	11/12/96
31	lubber	8	55.5	58	103	11/12/96

$$S1 \bowtie_{S1.sid < R1.sid} R1$$

- ***Result schema*** **same as that of cross-product.**
- **Fewer tuples than cross-product, usually able to compute more efficiently**
- **Sometimes called a *theta-join*.**

Joins

- ***Equi-Join***: **Special case: condition *c* contains only conjunction of** *equalities*.

sid	sname	rating	age	bid	day
22	dustin	7	45.0	101	10/10/96
58	rusty	10	35.0	103	11/12/96

$$S1 \bowtie_{sid} R1$$

- ***Result schema*** **similar to cross-product, but only one copy of fields for which equality is specified.**
- ***Natural Join***: **Equijoin on *all* common fields.**

Basic SQL

```
SELECT      [DISTINCT]  target-list
FROM        relation-list
WHERE       qualification
```

- **relation-list : A list of relation names**
  - possibly with a range-variable after each name
- **target-list : A list of attributes of tables in relation-list**
- **qualification : Comparisons combined using AND, OR and NOT.**
  - Comparisons are Attr op const or Attr1 op Attr2, where op is one of $< \;\; > \;\; = \;\; \leq \;\; \geq$
- **DISTINCT: optional keyword indicating that the answer should not contain duplicates.**
  - Default is that duplicates are not eliminated!

Conceptual Evaluation Strategy

- **Semantics of an SQL query defined in terms of the following conceptual evaluation strategy:**
  - Compute the cross-product of *relation-list*.
  - Discard resulting tuples if they fail *qualifications*.
  - Delete attributes that are not in *target-list*.
  - If DISTINCT is specified, eliminate duplicate rows.
- **Probably the least efficient way to compute a query!**
  - An optimizer will find more efficient strategies *same answers*.

Query Optimization & Processing

- **Optimizer maps SQL to algebra tree with specific algorithms**
  - access methods, join algorithms, scheduling
- **relational operators implemented as *iterators***
  - open()
  - next(possible with condition)
  - close
- **parallel processing engine built on partitioning dataflow to iterators**
  - inter- *and* intra-query parallelism

Workloads

- **Online Transaction Processing**
  - many little jobs (e.g. debit/credit)
  - SQL systems c. 1995 support 21,000 tpm-C
    - 112 cpu,670 disks
- **Batch (decision support and utility)**
  - few big jobs, parallelism inside
  - Scan data at 100 MB/s
  - Linear Scaleup to 500 processors

Today

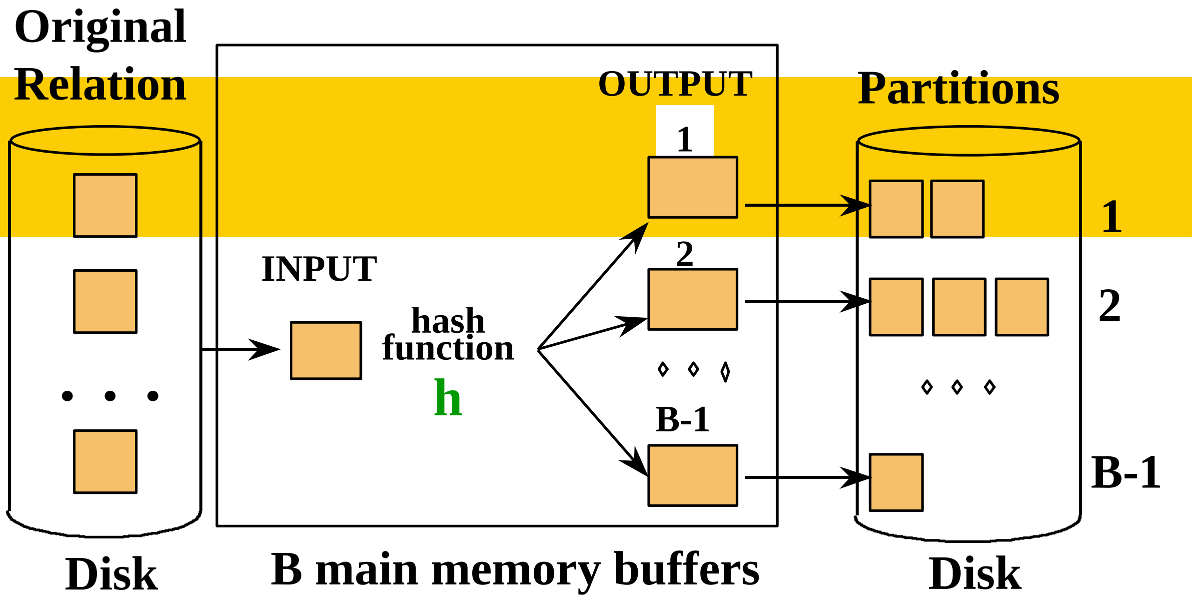
- **Background:**
  - The Relational Model and you
  - Meet a relational DBMS
- **Parallel Query Processing: sort and hash-join**
- **Data Layout**
- **Parallel Query Optimization**
- **Case Study: Teradata**

Parallelizing Sort

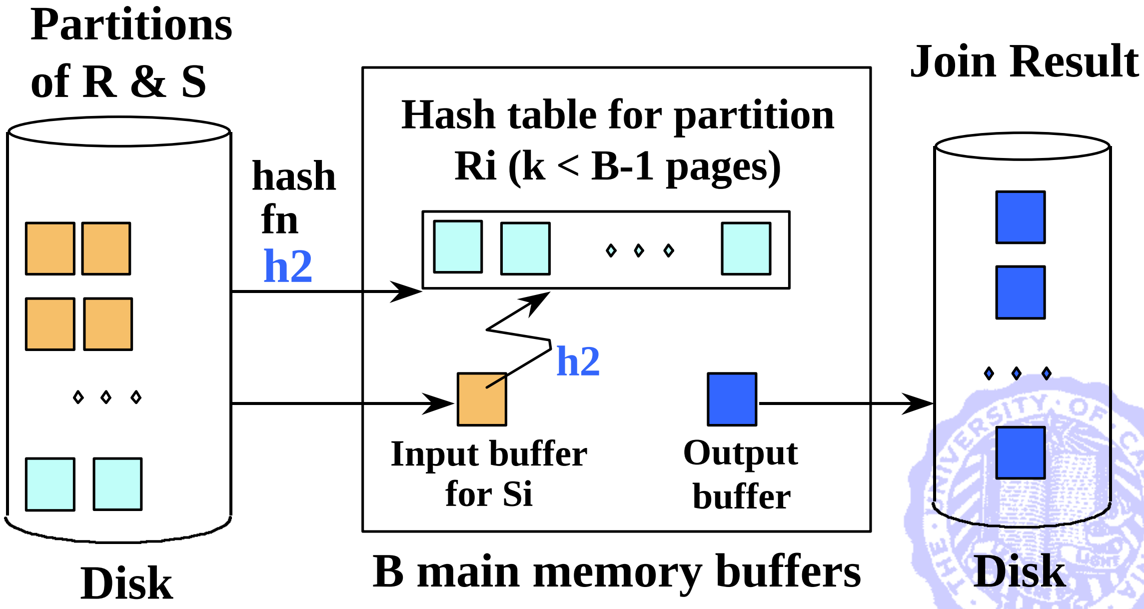
- **Why?**
  - DISTINCT, GROUP BY, ORDER BY, sort-merge join, index build
- **Phases:**
  - I: || read and partition (coarse radix sort), pipelined with || sorting of memory-sized runs, spilling runs to disk
  - || reading and merging of runs
- **Notes:**
  - phase 1 requires repartitioning 1-1/n of the data! High bandwidth network required.
  - phase 2 totally local processing
  - both pipelined *and* partitioned parallelism
  - *linear* speedup, scaleup!

Hash Join

- Partition both relations using hash fn h: R tuples in partition i will only match S tuples in partition i.



- ❖ Read in a partition of R, hash it using h2 (<> h!). Scan matching partition of S, search for matches.



Parallelizing Hash Join

- **Easy!**
  - Partition on join key in phase 1
  - Phase 2 runs locally

Themes in Parallel QP

- **essentially no synchronization except setup & teardown**
  - no barriers, cache coherence, etc.
  - DB transactions work fine in parallel
    - data updated in place, with 2-phase locking transactions
    - replicas managed only at EOT via 2-phase commit
    - coarser grain, higher overhead than cache coherency stuff
- **bandwidth much more important than latency**
  - often pump 1-1/n % of a table through the network
  - aggregate net BW should match aggregate disk BW
  - Latency, schmatency
- **ordering of data flow insignificant (hooray for relations!)**
  - Simplifies synchronization, allows for work-sharing
- **shared mem helps with skew**
  - but distributed work queues can solve this (?) (River)

Disk Layout

- **Where was the data to begin with?**
  - Major effects on performance
  - algorithms as described run at the speed of the slowest disk!
- **Disk placement**
  - logical partitioning, hash, round-robin
  - “declustering” for availability and load balance
  - indexes live with their data
- **This task is typically left to the “DBA”**
  - yuck!

Handling Skew

- **For range partitioning, sample load on disks.**
  - Cool hot disks by making range smaller
- **For hash partitioning,**
  - Cool hot disks by mapping some buckets to others

- **During query processing**
  - Use hashing and assume uniform
  - If range partitioning, sample data and use histogram to level the bulk
  - SMP/River scheme: work queue used to balance load

Query Optimization

- **Map SQL to a relational algebra tree, annotated with choice of algorithms. Issues:**
  - choice of access methods (indexes, scans)
  - join ordering
  - join algorithms
  - post-processing (e.g. hash vs. sort for groups, order)
- **Typical scheme, courtesy System R**
  - bottom-up dynamic-programming construction of entire plan space
  - prune based on cost and *selectivity* estimation

Parallel Query Optimization

- **More dimensions to plan space:**
  - degree of parallelism for each operator
  - scheduling: assignment of work to processors
- **One standard heuristic (Hong & Stonebraker)**
  - run the System R algorithm as if single-node (JOQR)
    - refinement: try to avoid repartitioning (query coloring)
  - parallelize (schedule) the resulting plan

Parallel Query Scheduling

- **Usage of a site by an isolated operator is given by *(T^{seq}, W, V)* where**
  - T^{seq} is the sequential execution time of the operator
  - *W* is a *d*-dimensional *work vector (time-shared)*
  - *V* is a *s*-dimensional *demand vector* (*space-shared*)
- **A set of "clones" $S = <(W_1,V_1),\ldots,(W_k,V_k)>$ is called *compatible* if they can be executed together on a site (space-shared constraint)**
- **Challenges:**
  - capture dependencies among operators (simple)
  - pick a degree of parallelism for each op (# of clones)
  - schedule clones to sites, under constraint of compatibility
- **solution is a mixture of query plan understanding, approximation algs for bin-packing, & modifications of dynamic programming optimization algs**

Today

- **Background:**
  - The Relational Model and you
  - Meet a relational DBMS
- **Parallel Query Processing: sort and hash-join**
- **Data Layout**
- **Parallel Query Optimization**
- **Case Study: Teradata**

Case Study: Teradata

- **Founded 1979: hardware and software**
  - beta 1982, shipped 1984
  - classic shared-nothing system
- **Hardware**
  - COP (Communications Processor)
    - accept, “plan”, “manage” queries
  - AMP (Access Module Processor)
    - SQL DB machine (own data, log, locks, executor)
    - Communicates with other AMPs directly
  - Ynet (now BYNET)
    - duplexed network (fault tolerance) among all nodes
    - sorts/merges messages by key
    - messages sent to all (Ynet routes hash buckets)
    - reliable multicast to groups of nodes
    - flow control via AMP pushback

History and Status

- **Bought by NCR/AT&T 1992**
- **AT&T spun off NCR again 1997**
- **TeraData software lives**
  - Word on the street: still running 8-bit PASCAL code
- **NCR WorldMark is the hardware platform**
  - Intel-based UNIX workstations + high-speed interconnect (a la IBM SP-2)
- **World's biggest online DB (?) is in TeraData**
  - Wal-Mart's sales data: 7.5 Tb on 365 AMPs

TeraData Data Layout

- **Hash everything**
  - All tables hash to 64000 buckets (64K in new version).
  - bucket map that distributes it over AMPS
- **AMPS manage local disks as one logical disk**
- **Data partitioned by primary index (may not be unique)**
  - Secondary indices too -- if unique, partitioned by key
  - if not unique, partitioned by hash of primary key
- **Fancy disk layout**
  - Key thing is that need for reorg is RARE (system is self organizing)
    - Occasionally run disk compaction (which is purely local)
    - Very easy to design and manage.

TeraData Query Execution

- **Complex queries executed "operator at a time",**
  - no pipelining between AMPs, some inside AMPS
- **Protocol**
  - 1. COP requests work
  - 2. AMPs all ACK starting (if not then backoff)
  - 3. get completion from all AMPs
  - 4. request answer (answers merged by Ynet)
  - 5. if it is a transaction, Ynet is used for 2-phase commit
- **Unique secondary index lookup:**
  - key->secondaryAMP->PrimaryAMP->ans
- **Non-Unique lookup:**
  - broadcast to all AMPs and then merge results

More on TeraData QP

- **MultiStatement operations can proceed in parallel (up to 10x parallel)**
  - e.g. batch of inserts or selects or even TP
- **Some intra-statement operators done in parallel**
  - E.g. (select * from x where ... order by ...) is three phases: scan->sort->spool->merge-> application.
  - AMP sets up a scanner, "catcher", and sorter
  - scanner reads records and throws qualifying records to Ynet (with hash sort key)
  - catcher gets records from Ynet and drives sorter
  - sorter generates locally sorted spool files.
  - when done, COP and Ynet do merge.
- **If join tables not equi-partitioned then rehash.**
  - Often replicate small outer table to many partitions (Ynet is good for this)

Lessons to Learn

- **Raising the abstraction to programmers is good!**
  - Allows advances in parallelization to proceed independently
- **Ordering, pointers and other structure are bad**
  - sets are great! partitionable without synch.
  - files have been a dangerous abstraction (encourage array-think)
  - pointers stink…think joins (same thing in batch!)
- **Avoiding low-latency messaging is a technology win**
  - shared-nothing clusters instead of MPP
  - Teradata lives, CM-5 doesn't…
  - UltraSparc lives too…CLUMPS

More Lessons

- **“Embarassing”?**
  - Perhaps, algorithmically
  - but ironed out a ton of HW/SW architectural issues
    - got interfaces right
    - iterators, dataflow, load balancing
    - building balanced HW systems
  - huge application space, big success
  - matches (drives?) the technology curve
    - linear speedup with better I/O interconnects, higher density and BW from disk
    - faster machines won’t make data problems go away

Moving Onward

- **Parallelism and Object-Relational**
  - can you give back the structure and keep the ||-ism?
  - E.g. multi-d objects, lists and array data, multimedia (usually arrays)
  - typical tricks include chunking and clustering, followed by sorting
    - I.e. try to apply set-like algorithms and “make right” later
  - lessons here?

History & Resources

- **Seminal research projects**
  - Gamma (DeWitt & co., Wisconsin)
  - Bubba (Boral, Copeland & Kim, MCC)
  - XPRS (Stonebraker & co, Berkeley)
  - Paradise? (DeWitt & co., Wisconsin)
- **Readings in Database Systems (CS286 text)**
  - http://redbook.cs.berkeley.edu
- **Jim Gray's Berkeley book report**
  - http://www.research.microsoft.com/~gray/PDB95.{doc,ppt}
- **Undergrad texts**
  - Ramakrishnan's "Database Management Systems"
  - Korth/Silberschatz/Sudarshan's "Database Systems Concepts"