Database Systems Architecture
DBMS Timeline

File-Based Applications
- Ad-hoc
- Efficient
- Un-manageable

Hierarchical / Network Systems
- Imperative
- Navigational

Relational Systems
- Data Independence
- Transaction support
- New features
- System R, Oracle, DB2, SqlServer, Postgres, MySQL, ...

Stream Query Processing

Analytics (Map-reduce)

OLTP (main-memory store)

OLAP (Column stores)
A **System** is a set of interconnected components that has an expected behaviour observed at the interface with its environment.

Coping with complexity:

- Modularity
- Abstractions
- Layering
- Hierarchy
- Iteration
Agenda

• Single Site DBMS
  – Process architecture
    • Single Core
    • Multi Core
  – Row vs. column store
  – Stream query processing

• Multi Site DBMS
  • Parallel DBMS
  • BigTable, H-base (Map-Reduce)
DBMS Components

- Parser
- Compiler
- Execution Engine
- Indexes
- Concurrency Control
- Buffer Manager
- Query Processor
- Storage Subsystem
- Recovery
DB2 Process Architecture
DBMS can run MUCH faster if they use new hardware efficiently
Staged DB

http://dias.epfl.ch/page74932.html
Contention in the lock manager

- Centralized service
  - Locks managed globally
- Fine-grained parallelism
  - Each lock has its own latch
- Skewed access
  - Some hotter than others

Culprit: shared high-level locks

Agenda

• Single Site DBMS
  – Process architecture
    • Single Core
    • Multi Core
  – Row vs. column store
  – Stream query processing

• Multi Site DBMS
  • Parallel DBMS
  • BigTable, H-base (Map-Reduce)
Row Store vs. Column Store

Row Database stores row values together

<table>
<thead>
<tr>
<th>EmpNo</th>
<th>EName</th>
<th>Job</th>
<th>Mgr</th>
<th>HireDate</th>
</tr>
</thead>
<tbody>
<tr>
<td>7369</td>
<td>SMITH</td>
<td>CLERK</td>
<td>7902</td>
<td>17/12/2000</td>
</tr>
<tr>
<td>7499</td>
<td>ALLEN</td>
<td>SALESMAN</td>
<td>7698</td>
<td>20/02/2001</td>
</tr>
<tr>
<td>7521</td>
<td>WARD</td>
<td>SALESMAN</td>
<td>7698</td>
<td>22/02/2001</td>
</tr>
<tr>
<td>7566</td>
<td>JONES</td>
<td>MANAGER</td>
<td>7839</td>
<td>2/04/1981</td>
</tr>
<tr>
<td>7654</td>
<td>MARTIN</td>
<td>SALESMAN</td>
<td>7698</td>
<td>28/09/1981</td>
</tr>
<tr>
<td>7698</td>
<td>BLAKE</td>
<td>MANAGER</td>
<td>7839</td>
<td>1/05/1981</td>
</tr>
<tr>
<td>7782</td>
<td>CLARK</td>
<td>MANAGER</td>
<td>7839</td>
<td>9/06/1981</td>
</tr>
</tbody>
</table>

Column Database stores column values together

<table>
<thead>
<tr>
<th>Block 1</th>
<th>Block 2</th>
<th>Block 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>7369</td>
<td>SMITH</td>
<td>CLERK</td>
</tr>
<tr>
<td>7499</td>
<td>ALLEN</td>
<td>SALESMAN</td>
</tr>
<tr>
<td>7521</td>
<td>WARD</td>
<td>SALESMAN</td>
</tr>
<tr>
<td>7566</td>
<td>JONES</td>
<td>MANAGER</td>
</tr>
<tr>
<td>7654</td>
<td>MARTIN</td>
<td>SALESMAN</td>
</tr>
<tr>
<td>7698</td>
<td>BLAKE</td>
<td>MANAGER</td>
</tr>
<tr>
<td>7782</td>
<td>CLARK</td>
<td>MANAGER</td>
</tr>
</tbody>
</table>
Vertical Partitioning

R(a, b, c) \rightarrow R(a,b), R(a,c)

Lossless decomposition iff
FD: a→b,c
or a key of R

Column store vs. Vertical partitioning:
• XXX
Agenda

• Single Site DBMS
  – Process architecture
    • Single Core
    • Multi Core
  – Row vs. column store
  – Stream query processing

• Multi Site DBMS
  • Parallel DBMS
  • BigTable, H-base (Map-Reduce)
Streaming Query Engine

Data → DBMS

Q → DBMS

Result Set
Streams

- A stream can be defined as a relation with a time stamp associated to each tuple.
  - Streams are list of tuples ordered by their time stamp

- Query language is SQL extended with a few extensions (system specific):
  - Window queries
  - Time rollup
  - Regular output

```
CREATE STREAM trades (symbol varchar(5),
  price real,
  volume integer,
  tstamp timestamp CQTIME USER
  GENERATED SLACK '1 minute'
) TYPE UNARCHIVED;
```

```
SELECT sum(price * volume) / sum(volume) AS vwap,
  sum(volume) AS volume,
  advance_agg(qtime) AS windowtime
FROM trades < VISIBLE '1 minute' ADVANCE '5 seconds' >
WHERE symbol = 'MSFT'
```
Stream Query Example

```
SELECT T.symbol, sum(T.price * T.volume)
FROM s_and_p_500 S,
    trades T < VISIBLE '5 sec' ADVANCE '3 sec' >
WHERE T.symbol = S.symbol
AND T.volume > 5000
GROUP BY T.symbol
```
Push vs. Pull Processing

Dimensions are largely orthogonal – all combinations are potentially useful.

Slide courtesy of M.Franklin (VLDB Tutorial 2001)
Stream Engine vs. RDBMS

- No time wasted loading data
- Windows vs. Set
  - Not all data have same importance
  - Smaller working set
- Multiple query executed on same data
- No need for transactions

- CEP: Complex Event Processing
  - Fraud detection
  - Trading
  - Business Activity Monitoring
- ETL: Extract, transform, Load
Agenda

• Single Site DBMS
  – Process architecture
    • Single Core
    • Multi Core
  – Row vs. column store
  – Stream query processing

• Multi Site DBMS
  • Parallel DBMS
  • BigTable, H-base (Map-Reduce)
Parallelism 101

• Step 1:
  − partition the work

• Step 2:
  − units of work processed in parallel

• Step 3:
  − report result

• Step 1 Problems:
  − Partition/replicate the data
  − Split/cloned processing
Partitioning

- Data can be distributed by storing individual tables at different sites.
- Data can also be distributed by decomposing a table and storing portions at different sites – called *partitioning*.
- Partitioning can be *horizontal* or *vertical*.
Horizontal Partitioning

- Each partition, $T_i$, of table $T$ contains a subset of the rows and each row is in exactly one partition.
Partitioning Techniques

Round-robin (n sites):
Send the $i^\text{th}$ tuple inserted in the relation to site $i \mod n$.

Hash partitioning:
- Choose one or more attributes as the partitioning attributes.
- Choose hash function $h$ with range $0…n - 1$
- Let $i$ denote result of hash function $h$ applied to the partitioning attribute value of a tuple. Send tuple to site $i$. 

Partitioning Techniques

Range partitioning:

• Choose an attribute as the partitioning attribute.
• A partitioning vector \([v_0, v_1, ..., v_{n-2}]\) is chosen.
• Let \(v\) be the partitioning attribute value of a tuple. Tuples such that \(v_i < v_{i+1}\) go to site \(l + 1\). Tuples with \(v < v_0\) go to disk 0 and tuples with \(v > v_{n-2}\) go to disk \(n-1\).
Partitioning Techniques

- Evaluate how well partitioning techniques support the following types of data access:
  1. Scanning the entire relation.
  2. Locating a tuple associatively – point queries.
     - E.g., \( r.A = 25 \).
  3. Locating all tuples such that the value of a given attribute lies within a specified range – range queries.
     - E.g., \( 10 < r.A < 25 \).
Round robin:

• **Advantages**
  
  • Best suited for sequential scan of entire relation on each query.
  
  • All disks have almost an equal number of tuples; retrieval work is thus well balanced between disks.

• **Range queries are difficult to process**
  
  • No clustering -- tuples are scattered across all disks
Partitioning Techniques

Hash partitioning:

- Good for sequential access
  - Assuming hash function is good, and partitioning attributes form a key, tuples will be equally distributed between sites
  - Retrieval work is then well balanced between sites.

- Good for point queries on partitioning attribute.
  Can lookup single site, leaving others available for answering other queries.

- No clustering, so difficult to answer range queries.
Partitioning Techniques

Range partitioning:
• Provides data clustering by partitioning attribute value.
• Good for sequential access
• Good for point queries on partitioning attribute: only one disk needs to be accessed.

• For range queries on partitioning attribute, one to a few disks may need to be accessed
• For range queries on non partitioning attribute, randomized access to sites.
Horizontal Partitioning

• *Example*: An Internet grocer has a relation describing inventory at each warehouse

  \[\text{Inventory}(\text{StockNum}, \text{Amount}, \text{Price}, \text{Location})\]

• It partitions the relation by location and stores each partition locally: rows with \(\text{Location} = \text{‘Chicago’}\) are stored in the Chicago warehouse in a partition

  \[\text{Inventory}_\text{ch}(\text{StockNum}, \text{Amount}, \text{Price}, \text{Location})\]

• Alternatively, it can use the schema

  \[\text{Inventory}_\text{ch}(\text{StockNum}, \text{Amount}, \text{Price})\]
Vertical Partitioning

• Each partition, $T_i$, of $T$ contains a subset of the columns, each column is in at least one partition, and each partition includes the key:
  
  $T_i = \pi_{\text{attr_list}_i}(T)$

  $T = T_1 \bowtie T_2 \ldots \bowtie T_n$

  - Vertical partitioning is lossless

• Example: The Internet grocer has a relation
  
  Employee($SSnum, Name, Salary, Title, Location$)

  - It partitions the relation to put some information at headquarters and some elsewhere:

  Emp1($SSnum, Name, Salary$) – at headquarters
  Emp2($SSnum, Name, Title, Location$) – elsewhere
Replication

• One of the most useful mechanisms in distributed databases
• Increases
  − Availability
    • If one replica site is down, data can be accessed from another site
  − Performance:
    • Queries can be executed more efficiently because they can access a local or nearby copy
    • Updates might be slower because all replicas must be updated
Replication Example

- Internet grocer might have relation
  \textbf{Customer}(\textit{CustNum, Address, Location})
  - Queries are executed
    - At headquarters to produce monthly mailings
    - At a warehouse to obtain information about deliveries
  - Updates are executed
    - At headquarters when new customer registers and when information about a customer changes
Example (con’t)

• Intuitively it seems appropriate to either or both:
  − Store complete relation at headquarters
  − Horizontally partition a replica of the relation and store a partition at the corresponding warehouse site

• Each row is replicated: one copy at headquarters, one copy at a warehouse

• The relation can be both distributed and replicated
Example (con’t): Performance Analysis

• We consider three alternatives:
  − Store the entire relation at the headquarters site and nothing at the warehouses (no replication)
  − Store the partitions at the warehouses and nothing at the headquarters (no replication)
  − Store entire relation at headquarters and a partition at each warehouse (replication)
Example (con’t):
Performance Analysis - Assumptions

• To evaluate the alternatives, we estimate the amount of information that must be sent between sites.
• Assumptions:
  − The Customer relation has 100,000 rows
  − The headquarters mailing application sends each customer 1 mailing a month
  − 500 deliveries are made each day; a single row is read for each delivery
  − 100 new customers/day
  − Changes to customer information occur infrequently
Example: The Evaluation

• **Entire relation at headquarters, nothing at warehouses**
  − 500 tuples per day from headquarters to warehouses for deliveries

• **Partitions at warehouses, nothing at headquarters**
  − 100,000 tuples per month from warehouses to headquarters for mailings (3,300 tuples per day, amortized)
  − 100 tuples per day from headquarters to warehouses for new customer registration

• **Entire relation at headquarters, partitions at warehouses**
  − 100 tuples per day from headquarters to warehouses for new customer registration
Example: Conclusion

• Replication (case 3) seems best, if we count the number of transmissions.
• Let us look at other measures:
  − If no data stored at warehouses, the time to handle deliveries might suffer because of the remote access (probably not important)
  − If no data is stored at headquarters, the monthly mailing requires that 100,000 rows be transmitted in a single day, which might clog the network
  − If we replicate, the time to register a new customer might suffer because of the remote update
    • But this update can be done by a separate transaction after the registration transaction commits (asynchronous update)
Distributed Parallelism 101

- **Step 1:**
  - partition the work

- **Step 2:**
  - units of work processed in parallel

- **Step 3:**
  - report result

- **Step 2 Problems:**
  - Assign work to processing unit
  - **Synchronization across processing units**
  - Aggregate result
  - Deal with failure
Parallelism Patterns

• Master / Workers
  - Master dispatches work to workers
  - Master collects sub-results from workers

• Producer / Consumers
  - Producers create work
  - Consumers process work
  - Producer-consumer mapping
    • 1-1: Network of producers / consumers
    • N-M: via a shared work queue
Example

• Consider the problem of feature extraction from a large data set:
  - For example: find the number of occurrences of a given word in a large data set (terabytes)

• Approach:
  - Partition data set across 100s/1000s of CPU
  - Parallelize word count
    • Dispatch work, aggregate result, deal with failure
Gamma Approach

- DBMS that supports Intra query parallelism
  - Pipelined and partitionned execution plans
The MapReduce Approach

• Run-time engine + Programming model
  - Automatic parallelization & distribution
  - Fault-tolerant
  - Provides status and monitoring tools
  - Clean abstraction for programmers
Map

• map :: (a -> b) -> [a] -> [b]
  map f [] = []
  map f (x:xs) = f x : map f xs

• Creates a new list by applying f to each element of the input list; returns output in order.
Programming Model

• Borrows from functional programming
• Users implement interface of two functions:
  - `map` (in_key, in_value) ->
    (out_key, intermediate_value) list
  - Records from the data source (lines out of files, rows of a database, etc) are fed into the map function as key*value pairs: e.g., (filename, line).
  - map() produces one or more intermediate values along with an output key from the input.
Programming Model

- reduce (out_key, intermediate_value list) -> out_value list
- After the map phase is over, all the intermediate values for a given output key are combined together into a list
- reduce() combines those intermediate values into one or more final values for that same output key
- (in practice, usually only one final value per key)
Example

map(String input_key, String input_value):

    // input_key: document name
    // input_value: document contents
    for each word w in input_value:
        EmitIntermediate(w, "1");

reduce(String output_key, Iterator intermediate_values):

    // output_key: a word
    // output_values: a list of counts
    int result = 0;
    for each v in intermediate_values:
        result += ParseInt(v);
    Emit(AsString(result));
Other examples

• need to count # of times every 5-word sequence occurs in large corpus of documents (and keep all those where count >= 4)

• With MapReduce:
  - map: extract 5-word sequences => count from document
  - reduce: combine counts, and keep if count large enough
Other Examples

• Example: generate per-doc summary, but include per-host information (e.g. # of pages on host, important terms on host)
  − per-host information might be in per-process data structure, o might involve RPC to a set of machines containing data for all sites
• map: extract host name from URL, lookup per-host info
• combine with per-doc data and emit)
MapReduce Parallelism

- **map()** functions run in parallel, creating different intermediate values from different input data sets
- **reduce()** functions also run in parallel, each working on a different output key
- All values are processed independently
- **Bottleneck**: reduce phase can’t start until map phase is completely finished.
Map + Reduce

Input key *value pairs

Data store 1

map

(key 1, values ...)
(key 2, values ...)
(key 3, values ...)

map

Data store n

(key 1, values ...)
(key 2, values ...)
(key 3, values ...)

== Barrier == : Aggregates intermediate values by output key

key 1, intermediate values
reduce
final key 1 values

key 2, intermediate values
reduce
final key 2 values

key 3, intermediate values
reduce
final key 3 values
Execution Engine
Failures

• Master detects worker failures
  − Re-executes completed & in-progress map() tasks
  − Re-executes in-progress reduce() tasks

• Master notices particular input key/values cause crashes in map(), and skips those values on re-execution.
  − Effect: Can work around bugs in third-party libraries!
Optimizations

• No reduce can start until map is complete:
  - A single slow disk controller can rate-limit the whole process
• Master redundantly executes “slow-moving” map tasks; uses results of first copy to finish
Map Reduce vs. DBMS

• Pipelined vs. Staged execution
  – Staged execution less performant (in the absence of failure), better fault-tolerance.
  – 100s vs 10000s of nodes

• Start-up vs. Running cost
  – Map-reduce has no start-up cost (parsing text data is overhead – data stored in binary format – google protocol buffers)

  – In DBMS cost of loading data amortized over many queries.
DBMS over MapReduce

- H-Base:
  - http://hadoop.apache.org/hbase/

- BigTable
  - http://video.google.com/videoplay?docid=7278544055668715642#