Big Data

Donald Kossmann & Nesime Tatbul
Systems Group
ETH Zurich
Goal of this Module

• Understand how Big Data has been done so far
  – i.e., how to exploit relational database systems
  – which data models to use
  – some interesting algorithms

• Also, understand the limitations and why we need new technology
  – you need to understand the starting point!
Puzzle of the Day

• There is a jazz festival in Montreux.
• Make sure Migros Montreux has enough beer.

• This is a Big Data problem!
  – how much beer do we need in each store?
• How does Migros solve that problem today?
  – data warehouses (today)
• How could Migros solve that problem in future?
  – data warehouses + event calendar + Facebook + ...
  – (coming weeks)
Selected References on Data Warehouses

• General
  – Chaudhuri, Dayal: An Overview of Data Warehousing and OLAP Technology. SIGMOD Record 1997
  – Lehner: Datenbanktechnologie für Data Warehouse Systeme. Dpunkt Verlag 2003
  – (…)

• New Operators and Algorithms
  – Agrawal, Srikant: Fast Algorithms for Association Rule Mining. VLDB 1994
  – Börszonyi, Kossmann, Stocker: Skyline Operator. ICDE 2001
  – Carey, Kossmann: On Saying Enough Already in SQL. SIGMOD 1997
  – Gray et al.: Data Cube... ICDE 1996
  – (…)

History of Databases

• Age of Transactions (70s - 00s)
  – Goal: reliability - make sure no data is lost
  – 60s: IMS (hierarchical data model)
  – 80s: Oracle (relational data model)

• Age of Business Intelligence (95 -)
  – Goal: analyze the data -> make business decisions
  – Aggregate data for boss. Tolerate imprecision!
  – SAP BW, Microstrategy, Cognos, … (rel. model)

• Age of „Big Data“ and „Data for the Masses“
  – Goal: everybody has access to everything, M2M
  – Google (text), Cloud (XML, JSON: Services)
Some Selected Topics

- Motivation and Architecture
- SQL Extensions for Data Warehousing (DSS)
- Algorithms and Query Processing Techniques
- ETL, Virtual Databases (Data Integration)
- Parallel Databases
- Column Stores, Vector Databases
- Data Mining
- Probabilistic Databases
- Temporal Databases

- This is a whole class for itself (Spring semester)
  - we will only scratch the surface here
OLTP vs. OLAP

• OLTP – Online Transaction Processing
  – Many small transactions
    (point queries: UPDATE or INSERT)
  – Avoid redundancy, normalize schemas
  – Access to consistent, up-to-date database

• OLTP Examples:
  – Flight reservation (see IS-G)
  – Order Management, Procurement, ERP

• Goal: 6000 Transactions per second (Oracle 1995)
OLTP vs. OLAP

• OLAP – Online Analytical Processing
  – Big queries (all the data, joins); no Updates
  – Redundancy a necessity (Materialized Views, special-purpose indexes, de-normalized schemas)
  – Periodic refresh of data (daily or weekly)

• OLAP Examples
  – Management Information (sales per employee)
  – Statistisches Bundesamt (Volkszählung)
  – Scientific databases, Bio-Informatics

• Goal: Response Time of seconds / few minutes
OLTP vs. OLAP (Water and Oil)

- **Lock Conflicts:** OLAP blocks OLTP
- **Database design:**
  - OLTP normalized, OLAP de-normalized
- **Tuning, Optimization**
  - OLTP: inter-query parallelism, heuristic optimization
  - OLAP: intra-query parallelism, full-fledged optimization
- **Freshness of Data:**
  - OLTP: serializability
  - OLAP: reproducability
- **Precision:**
  - OLTP: ACID
  - OLAP: Sampling, Confidence Intervals
Solution: Data Warehouse

- Special Sandbox for OLAP
- Data input using OLTP systems
- Data Warehouse aggregates and replicates data (special schema)
- New Data is *periodically* uploaded to Warehouse
- Old Data is deleted from Warehouse
  - Archiving done by OLTP system for legal reasons
Architecture

OLTP

OLTP Applications

DB1

DB2

DB3

OLAP

GUI, Spreadsheets

Data Warehouse
Limitations of State of the Art

Business Processes

Manual Analysis

does not scale

Storage Network

ETL into RDBMS

inflexible + data loss

data is dead

Archive

does not scale
Data Warehouses in the real World

• First industrial projects in 1995
• At beginning, 80% failure rate of projects
• Consultants like Accenture dominate market
• Why difficult: Data integration + cleaning, poor modeling of business processes in warehouse
• Data warehouses are expensive (typically as expensive as OLTP system)
• Success Story: WalMart - 20% cost reduction because of Data Warehouse (just in time...)
Products and Tools

• Oracle 11g, IBM DB2, Microsoft SQL Server, ...
  – All database vendors
• SAP Business Information Warehouse (Hana)
  – ERP vendors
• MicroStrategy, Cognos
  – Specialized vendors
  – „Web-based EXCEL“
• Niche Players (e.g., Btell)
  – Vertical application domain
ETL Process

• **Major Cost Factors of Data Warehousing**
  – define schema / data model (next)
  – define ETL process

• **ETL Process**
  – extract: suck out the data from OLTP system
  – transform: clense it, bring it into right format
  – load: add it to the data warehouse

• **Staging areas**
  – modern data warehouses keep results at all stages
Some Details

• **Extract**
  – easy, if OLTP is a relational database
    • (use triggers, replication facilities, etc.)
  – more difficult, if OLTP data comes from file system

• **Transform**
  – data cleaning: can be arbitrary complicated
    • machine learning, workflow with human input, ...
  – structures: many tools that generate code

• **Load**
  – use bulkloading tools from vendors
Some Considerations

• **When to ETL data?**
  – freshness: periodically vs. continuously
  – consistency: do we need to transact the ETLs

• **Granularity of ETL?**
  – individual tuples vs. batches
  – cost / freshness / quality tradeoffs
    • often a batch can be better cleaned

• **Infrastructure?**
  – ETL from same machine or even same DB
  – workload / performance separation vs. cost
ETL vs. Big Data

• ETL is the exact opposite of “modern” Big Data
  – “speed”: does not really work for fast data
  – philosophy: change question -> change ETL workflow

• Big Data prefers in-situ processing
  – “volume”: not all data is worth ETLing
  – “statistics”: error may be part of the signal (!)
  – “cost:” why bother if you can have it all in one
    • products like SAP Hana also go into this direction
  – “diversity:” increases complexity of ETL process

• But, Big Data has no magic with regard to quality
  – and ETL great if investment is indeed worth-while
    • valuable data vs. mass data
Star Schema (relational)

- Dimension Table (e.g., POS)
- Dimension Table (e.g., Customer)
- Dimension Table (e.g., Supplier)
- Dimension Table (e.g., Product)
- Dimension Table (e.g., Time)

Fact Table (e.g., Order)
## Fact Table (Order)

<table>
<thead>
<tr>
<th>No.</th>
<th>Cust.</th>
<th>Date</th>
<th>POS</th>
<th>Price</th>
<th>Vol.</th>
<th>TAX</th>
</tr>
</thead>
<tbody>
<tr>
<td>001</td>
<td>Heinz</td>
<td>13.5.</td>
<td>Mainz</td>
<td>500</td>
<td>5</td>
<td>7.0</td>
</tr>
<tr>
<td>002</td>
<td>Ute</td>
<td>17.6.</td>
<td>Köln</td>
<td>500</td>
<td>1</td>
<td>14.0</td>
</tr>
<tr>
<td>003</td>
<td>Heinz</td>
<td>21.6.</td>
<td>Köln</td>
<td>700</td>
<td>1</td>
<td>7.0</td>
</tr>
<tr>
<td>004</td>
<td>Heinz</td>
<td>4.10.</td>
<td>Mainz</td>
<td>400</td>
<td>7</td>
<td>7.0</td>
</tr>
<tr>
<td>005</td>
<td>Karin</td>
<td>4.10.</td>
<td>Mainz</td>
<td>800</td>
<td>3</td>
<td>0.0</td>
</tr>
<tr>
<td>006</td>
<td>Thea</td>
<td>7.10.</td>
<td>Köln</td>
<td>300</td>
<td>2</td>
<td>14.0</td>
</tr>
<tr>
<td>007</td>
<td>Nobbi</td>
<td>13.11.</td>
<td>Köln</td>
<td>100</td>
<td>5</td>
<td>7.0</td>
</tr>
<tr>
<td>008</td>
<td>Sarah</td>
<td>20.12</td>
<td>Köln</td>
<td>200</td>
<td>4</td>
<td>7.0</td>
</tr>
</tbody>
</table>
Fact Table

• **Structure:**
  – key (e.g., Order Number)
  – Foreign key to all dimension tables
  – measures (e.g., Price, Volume, TAX, …)

• **Store *moving data* (**Bewegungsdaten**)

• **Very large and normalized**
## Dimension Table (PoS)

<table>
<thead>
<tr>
<th>Name</th>
<th>Manager</th>
<th>City</th>
<th>Region</th>
<th>Country</th>
<th>Tel.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mainz</td>
<td>Helga</td>
<td>Mainz</td>
<td>South</td>
<td>D</td>
<td>1422</td>
</tr>
<tr>
<td>Köln</td>
<td>Vera</td>
<td>Hürth</td>
<td>South</td>
<td>D</td>
<td>3311</td>
</tr>
</tbody>
</table>

- **De-normalized**: City -> Region -> Country
  - Avoid joins
- fairly small and constant size
- Dimension tables store *master data (Stammdaten)*
- Attributes are called *Merkmale* in German
Snowflake Schema

• If dimension tables get too large
  – Partition the dimension table

• Trade-Off
  – Less redundancy (smaller tables)
  – Additional joins needed

• Exercise: Do the math!
Typical Queries

SELECT  d1.x, d2.y, d3.z, sum(f.z1), avg(f.z2)
FROM     Fact f, Dim1 d1, Dim2 d2, Dim3 d3
WHERE a < d1.feld < b  AND  d2.feld = c AND 
Join predicates
GROUP BY d1.x, d2.y, d3.z;

• Select by Attributes of Dimensions
  – E.g., region = „south“
• Group by Attributes of Dimensions
  – E.g., region, month, quarter
• Aggregate on measures
  – E.g., sum(price * volumen)
Example

```sql
SELECT  f.region, z.month, sum(a.price * a.volume)
FROM     Order a, Time z, PoS f
WHERE  a.pos = f.name AND a.date = z.date
GROUP BY f.region, z.month
```

<table>
<thead>
<tr>
<th>Region</th>
<th>Month</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>South</td>
<td>May</td>
<td>2500</td>
</tr>
<tr>
<td>North</td>
<td>June</td>
<td>1200</td>
</tr>
<tr>
<td>South</td>
<td>October</td>
<td>5200</td>
</tr>
<tr>
<td>North</td>
<td>October</td>
<td>600</td>
</tr>
</tbody>
</table>
Star Schema vs. Big Data

• **Star Schema designed for specific questions**
  – define “metrics” and “dimensions” upfront
  – thus, define questions you can ask upfront
  – great for operational BI
  – bad for ad-hoc questions (e.g., disasters)
  – breaks philosophy of Big Data (collect, then think)
    • e.g., health record: is “disease” metric or dimension?

• **Poor on diversity**
  – even if you know all the questions upfront, you may end up with multiple Star schemas
Drill-Down und Roll-Up

• Add attribute to GROUP BY clause
  – More detailed results (e.g., more fine-grained results)

• Remove attribute from GROUP BY clause
  – More coarse-grained results (e.g., big picture)

• GUIs allow „Navigation“ through Results
  – Drill-Down: more detailed results
  – Roll-Up: less detailed results

• Typical operation, drill-down along hierarchy:
  – E.g., use „city“ instead of „region“
Data Cube

Sales by Product and Year

Product

0

Balls

1

Nets

2

Region

North

South

all

Year

1998

1999

2000

All
Moving Sums, ROLLUP

• Example:
  GROUP BY ROLLUP(country, region, city)
  Give totals for all countries and regions

• This can be done by using the ROLLUP Operator

• Attention: The order of dimensions in the GROUP BY clause matters!!!

• Again: Spreadsheets (EXCEL) are good at this

• The result is a table! (Completeness of rel. model!)
ROLLUP alla IBM UDB

```sql
SELECT Country, Region, City, sum(price*vol)
FROM Orders a, PoS f
WHERE a.pos = f.name
GROUP BY ROLLUP(Country, Region, City)
ORDER BY Country, Region, City;
```

Also works for other aggregate functions; e.g., avg().
## Result of ROLLUP Operator

<table>
<thead>
<tr>
<th>D</th>
<th>North</th>
<th>Köln</th>
<th>1000</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td>North</td>
<td>(null)</td>
<td>1000</td>
</tr>
<tr>
<td>D</td>
<td>South</td>
<td>Mainz</td>
<td>3000</td>
</tr>
<tr>
<td>D</td>
<td>South</td>
<td>München</td>
<td>200</td>
</tr>
<tr>
<td>D</td>
<td>South</td>
<td>(null)</td>
<td>3200</td>
</tr>
<tr>
<td>D</td>
<td>(null)</td>
<td>(null)</td>
<td>4200</td>
</tr>
</tbody>
</table>
Summarizability (Unit)

- **Legal Query**
  ```sql
  SELECT product, customer, unit, sum(volume)
  FROM Order
  GROUP BY product, customer, unit;
  ```

- **Legal Query (product -> unit)**
  ```sql
  SELECT product, customer, sum(volume)
  FROM Order
  GROUP BY product, customer;
  ```

- **Illegal Query (add „kg“ to „m“)!!!**
  ```sql
  SELECT customer, sum(volume)
  FROM Order
  GROUP BY customer;
  ```
## Summarizability (de-normalized data)

<table>
<thead>
<tr>
<th>Region</th>
<th>Customer</th>
<th>Product</th>
<th>Volume</th>
<th>Populat.</th>
</tr>
</thead>
<tbody>
<tr>
<td>South</td>
<td>Heinz</td>
<td>Balls</td>
<td>1000</td>
<td>3 Mio.</td>
</tr>
<tr>
<td>South</td>
<td>Heinz</td>
<td>Nets</td>
<td>500</td>
<td>3 Mio.</td>
</tr>
<tr>
<td>South</td>
<td>Mary</td>
<td>Balls</td>
<td>800</td>
<td>3 Mio.</td>
</tr>
<tr>
<td>South</td>
<td>Mary</td>
<td>Nets</td>
<td>700</td>
<td>3 Mio.</td>
</tr>
<tr>
<td>North</td>
<td>Heinz</td>
<td>Balls</td>
<td>1000</td>
<td>20 Mio.</td>
</tr>
<tr>
<td>North</td>
<td>Heinz</td>
<td>Nets</td>
<td>500</td>
<td>20 Mio.</td>
</tr>
<tr>
<td>North</td>
<td>Mary</td>
<td>Balls</td>
<td>800</td>
<td>20 Mio.</td>
</tr>
<tr>
<td>North</td>
<td>Mary</td>
<td>Nets</td>
<td>700</td>
<td>20 Mio.</td>
</tr>
</tbody>
</table>

Customer, Product -> Revenue
Region -> Population
Summarizability (de-normalized data)

• What is the result of the following query?

```
SELECT region, customer, product, sum(volume)
FROM Order
GROUP BY ROLLUP(region, customer, product);
```

• All off-the-shelf databases get this wrong!
• Problem: Total Revenue is 3000 (not 6000!)
• BI Tools get it right: keep track of functional dependencies
• Problem arises if reports involve several unrelated measures.
Overview

• Motivation and Architecture

• **SQL Extensions for Data Warehousing (DSS)**
  – Algorithms and Query Processing Techniques

• Column Stores, Vector Databases

• Parallel Databases

• Operational BI
Cube Operator

- Operator that computes all "combinations"
- Result contains "(null)" Values to encode "all"

```sql
SELECT product, year, region, sum(price * vol)
FROM Orders
GROUP BY CUBE(product, year, region);
```
## Result of Cube Operator

<table>
<thead>
<tr>
<th>Product</th>
<th>Region</th>
<th>Year</th>
<th>Revenue</th>
</tr>
</thead>
<tbody>
<tr>
<td>Netze</td>
<td>Nord</td>
<td>1998</td>
<td>...</td>
</tr>
<tr>
<td>Bälle</td>
<td>Nord</td>
<td>1998</td>
<td>...</td>
</tr>
<tr>
<td>(null)</td>
<td>Nord</td>
<td>1998</td>
<td>...</td>
</tr>
<tr>
<td>Netze</td>
<td>Süd</td>
<td>1998</td>
<td>...</td>
</tr>
<tr>
<td>Bälle</td>
<td>Süd</td>
<td>1998</td>
<td>...</td>
</tr>
<tr>
<td>(null)</td>
<td>Süd</td>
<td>1998</td>
<td>...</td>
</tr>
<tr>
<td>Netze</td>
<td>(null)</td>
<td>1998</td>
<td>...</td>
</tr>
<tr>
<td>Bälle</td>
<td>(null)</td>
<td>1998</td>
<td>...</td>
</tr>
<tr>
<td>(null)</td>
<td>(null)</td>
<td>1998</td>
<td>...</td>
</tr>
</tbody>
</table>
Visualization as Cube

Product

all

Balls

Nets

Region

North  South  all

Year

1998  1999  2000

all
Computation Graph of Cube

{product, year, region}

{product}  {year}  {region}

{product, year}  {product, region}  {year, region}

{product, year, region}
Pivot Tables

• Define „columns“ by group by predicates
• Not a SQL standard! But common in products
• Reference:
  • Cunningham, Graefe, Galindo-Legaria: PIVOT and UNPIVOT: Optimization and Execution Strategies in an RDBMS. VLDB 2004
Materialized Views

• Compute the result of a query using the result of another query

• Principle: Subsumption
  – The set of all German researchers is a subset of the set of all researchers
  – If query asks for German researchers, use set of all researchers, rather than all people

• Subsumption works well for GROUP BY
Materialized View

SELECT product, year, region, sum(price * vol) 
FROM Order 
GROUP BY product, year, region;

GROUP BY product, year

SELECT product, year, sum(price * vol) 
FROM Order 
GROUP BY product, year;
Optimization of Group Bys

• Give each department with salary of employees
  \[ \text{SELECT e.dno, d.budget, \text{sum}(e.salary) FROM Emp e, Dept d WHERE e.dno = d.dno GROUP BY e.dno, d.budget;} \]

• Plan 1: Join before Group By (classic)
  – \( \Gamma(\text{Emp 1 Dept}) \)

• Plan 2: Join after Group By (advanced)
  – \( \Gamma(\text{Emp}) 1 \text{ Dept} \)

• Assessment
  – Why (or when) is Plan 2 legal?
  – Why (or when) is Plan 1 better than Plan 2?
UNPIVOT (material, factory)

<table>
<thead>
<tr>
<th>mat.code</th>
<th>mat.name</th>
<th>factory</th>
<th>vol. total</th>
<th>price [avg]</th>
</tr>
</thead>
<tbody>
<tr>
<td>APP</td>
<td>Apples</td>
<td>Berlin</td>
<td>32,000,000.00</td>
<td>0.44</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Dallas</td>
<td>36,800,000.00</td>
<td>0.28</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Duisburg</td>
<td>28,800,000.00</td>
<td>0.42</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Hanoi</td>
<td>24,000,000.00</td>
<td>0.20</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Madison</td>
<td>40,000,000.00</td>
<td>0.28</td>
</tr>
<tr>
<td>B 125</td>
<td>Bottles 125 ml</td>
<td>Hanoi</td>
<td>92,000.00</td>
<td>70.00</td>
</tr>
<tr>
<td>B 250</td>
<td>Bottle 250 ml</td>
<td>Hanoi</td>
<td>48,000.00</td>
<td>105.20</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Tokyo</td>
<td>36,000.00</td>
<td>87.50</td>
</tr>
<tr>
<td>BAN</td>
<td>Bananas</td>
<td>Hanoi</td>
<td>72,000,000.00</td>
<td>0.10</td>
</tr>
<tr>
<td>BEE</td>
<td>Ground Beef</td>
<td>Berlin</td>
<td>2,400,000.00</td>
<td>2.25</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Dallas</td>
<td>6,000,000.00</td>
<td>1.00</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Duisburg</td>
<td>900,000.00</td>
<td>2.25</td>
</tr>
<tr>
<td>BLA</td>
<td>Blackberries, dried</td>
<td>Madison</td>
<td>6,000.00</td>
<td>224.00</td>
</tr>
<tr>
<td>CAL</td>
<td>Calcium</td>
<td>Berlin</td>
<td>6,000.00</td>
<td>120.00</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Dallas</td>
<td>6,000.00</td>
<td>112.00</td>
</tr>
</tbody>
</table>
### Factory Report

- **Plan:** 2006 Update 1
- **Currency:** EUR
- **Computed at:** 04/02/2006 7:00 PM

#### Report

<table>
<thead>
<tr>
<th>mat.code</th>
<th>mat.name</th>
<th>Berlin</th>
<th>Dallas</th>
<th>Duisburg</th>
<th>Hanoi</th>
<th>Madison</th>
<th>Price [avg]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>vol. total</td>
<td>vol. total</td>
<td>vol. total</td>
<td>vol. total</td>
<td>vol. total</td>
<td>vol. total</td>
</tr>
<tr>
<td>APP</td>
<td>Apples</td>
<td>32,000,000.0</td>
<td>36,800,000.0</td>
<td>28,800,000.0</td>
<td>24,000,000.0</td>
<td>40,000,000.0</td>
<td>0.44</td>
</tr>
<tr>
<td>B 125</td>
<td>Bottles 125 ml</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B 250</td>
<td>Bottle 250 ml</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BAN</td>
<td>Bananas</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BEE</td>
<td>Ground Beef</td>
<td>2,400,000.0</td>
<td>6,000,000.0</td>
<td>900,000.0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BLA</td>
<td>Blackberries, dried</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CAL</td>
<td>Calcium</td>
<td>6,000.0</td>
<td>6,000.0</td>
<td>6,000.0</td>
<td>6,000.0</td>
<td>6,000.0</td>
<td></td>
</tr>
<tr>
<td>CAR</td>
<td>Carrots</td>
<td>15,600,000.0</td>
<td>13,200,000.0</td>
<td>14,400,000.0</td>
<td>6,000.0</td>
<td>6,000.0</td>
<td></td>
</tr>
<tr>
<td>CAU</td>
<td>Cauliflower</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CB 500</td>
<td>Cardboard Box 500g</td>
<td>60,000.0</td>
<td>60,000.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CHE</td>
<td>Cheese</td>
<td></td>
<td>1,200,000.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CHI</td>
<td>Ground Chicken</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CHO</td>
<td>Chocolate</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CIN</td>
<td>Cinnamon</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>COR</td>
<td>Corn</td>
<td>18,400,000.0</td>
<td>30,000,000.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Download EXCEL file  Download PDF file  Create Diagramm  Create bookmark

- **Add dimension (vertical):** none
- **Add dimension (horizontal):** none
- **Lines per page:** 15
Top N

- Many applications require top N queries
- Example 1 - Web databases
  - find the five cheapest hotels in Madison
- Example 2 - Decision Support
  - find the three best selling products
  - average salary of the 10,000 best paid employees
  - send the five worst batters to the minors
- Example 3 - Multimedia / Text databases
  - find 10 documents about “database” and “web”.
- Queries and updates, any N, all kinds of data
Key Observation

Top N queries cannot be expressed *well* in SQL

SELECT * FROM Hotels h
WHERE city = Madison AND
  5 > (SELECT count(*) FROM Hotels h1
       WHERE city = Madison AND h1.price < h.price);

• **So what do you do?**
  – Implement top N functionality in your application
  – Extend SQL and the database management system
Implementation of Top N in the App

• Applications use SQL to get as close as possible
• Get results *ordered*, consume only N objects and/or specify *predicate* to limit # of results

```
SELECT * FROM Hotels
WHERE city = Madison
ORDER BY price;
```

```
SELECT * FROM Hotels
WHERE city = Madison
AND price < 70;
```

– either too many results, **poor performance**
– or not enough results, **user must ask query again**
– difficult for nested top N queries and updates
Extend SQL and DBMS

- **STOP AFTER** clause specifies number of results

  ```sql
  SELECT         *
  FROM             Hotels
  WHERE         city = Madison
  ORDER BY     price
  STOP AFTER 5   [WITH TIES];
  ```

- Returns five hotels (plus ties)

- Challenge: extend query processor, performance
Updates

• Give top 5 salesperson a 50% salary raise

UPDATE Salesperson SET salary = 1.5 * salary
WHERE id IN
(SELECT id
FROM Salesperson
ORDER BY turnover DESC
STOP AFTER 5);
Nested Queries

• The average salary of the top 10000 Emps

```
SELECT AVG(salary)  
FROM (SELECT salary   
      FROM Emp      
      ORDER BY salary DESC   
      STOP AFTER 10000);  
```
Extend SQL and DBMSs

• SQL syntax extension needed
• All major database vendors do it
• Unfortunately, everybody uses a different syntax
  – Microsoft:  set rowcount $N$
  – IBM DB2:  fetch first $N$ rows only
  – Oracle:  rownum < $N$ predicate
  – SAP R/3:  first $N$

• Challenge: extend query processor of a DBMS
Top N Queries Revisited

• Example: The five cheapest hotels
  
  ```sql
  SELECT *
  FROM Hotels
  ORDER BY price
  STOP AFTER 5;
  ```

• What happens if you have several criteria?
Nearest Neighbor Search

• Cheap and close to the beach

```
SELECT * 
FROM Hotels 
ORDER BY distance * x + price * y 
STOP AFTER 5;
```

• How to set $x$ and $y$ ?
SELECT * 
FROM Hotels h, Cities c 
WHERE h.city = c.name;

Parser & Query Optimizer

Execution Engine

Scan(Hotels)

Scan(Cities)

Hash Join

<Ritz, Paris, ...>
<brisser Hase, Passau, ...>
<Edgewater, Madison, ...>

Catalogue

Indexes & Base Data
Processing Top N Queries

• Overall goal: avoid wasted work

• **Stop** operators encapsulate **top N** operation
  – implementation of other operators not changed

• Extend optimizer to produce plans with **Stop**

```
SELECT *
FROM Hotels h, Cities c
WHERE h.city = c.name
ORDER BY h.price
STOP AFTER 10;
```
Implementation of Stop Operators

• Several alternative ways to implement Stop
• Performance depends on:
  – N
  – availability of indexes
  – size of available main memory
  – properties of other operations in the query
Implementation Variants

• **Stop after a Sort** (trivial)

• **Priority queue**
  – build main memory priority queue with first N objects of input
  – read other objects one at a time:
    test membership bounds & replace

• **Partition** the input (range-based braking)

• **Stop after an Index-Scan**
Range-based Braking

– Adapt ideas from parallel sorting [DeWitt&Naughton]
– Use histograms (if available) or sampling
Range-based Braking Variants

1. **Materialize:** store all partitions on disk

   - Input
     - scan
     - sort
   - restart
     - stop

2. **Reread:** scan input for each partition

   - Input
     - scan
     - filter
     - sort
   - restart
     - stop

3. **Hybrid:** materialize first x partitions; reread others
# Performance of Stop Operators

N highest paid Emps; AODB/Sun; 4 MB mem.; 50 MB DB

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>10</td>
<td>100</td>
<td>50K</td>
<td>ALL</td>
</tr>
<tr>
<td>Sort</td>
<td>104.2</td>
<td>103.2</td>
<td>112.2</td>
<td>117.9</td>
</tr>
<tr>
<td>PQ</td>
<td>54.0</td>
<td>52.7</td>
<td>n.a.n</td>
<td>n.a.n</td>
</tr>
<tr>
<td>Mat</td>
<td>75.3</td>
<td>75.0</td>
<td>83.6</td>
<td>120.1</td>
</tr>
<tr>
<td>Reread</td>
<td>50.0</td>
<td>50.1</td>
<td>83.6</td>
<td>120.1</td>
</tr>
<tr>
<td>Hybrid</td>
<td>49.5</td>
<td>50.0</td>
<td>87.6</td>
<td>126.4</td>
</tr>
</tbody>
</table>
Stop & Indexes

- Read and follow pointers from index until N results have been produced
- Very simple to implement, result is sorted
- Random I/O if N is large or if there is an additional predicate (e.g., hotels in Madison)
• read first partition
• sort pointers to avoid random I/O
• read objects using (sorted) pointers
• re-sort tuples
• repeat until N results are produced
## Performance Evaluation (Index)

N highest paid Emps; AODB/Sun; 4 MB mem.; 50 MB DB

<table>
<thead>
<tr>
<th></th>
<th>N=10</th>
<th>N=1K</th>
<th>N=10K</th>
<th>N=50K</th>
</tr>
</thead>
<tbody>
<tr>
<td>Index</td>
<td>1.8</td>
<td>92.8</td>
<td>807.4</td>
<td>4505.5</td>
</tr>
<tr>
<td>Part&amp;Index</td>
<td>1.0</td>
<td>7.8</td>
<td>31.0</td>
<td>148.1</td>
</tr>
<tr>
<td>Hybrid</td>
<td>49.5</td>
<td>55.0</td>
<td>55.7</td>
<td>87.6</td>
</tr>
</tbody>
</table>
Optimizing Top N Queries

• Traditional optimizer must decide
  – join order
  – access paths (i.e., use of indexes), ...

• Top N optimizer must in addition decide
  – which implementation of Stop operator to use
  – where in a plan to place Stop operators

• Optimizer enumerates all alternative plans and selects best plan using a cost model

• Stop operators affect other decisions
  (e.g., join order, access paths)
Favor Pipelined Plans for Small N

- **pipelining operators** process a tuple at a time
  - idxscan (hotels)
  - NL Join
  - Stop

- **blocking operators** consume whole input
  - scan (cities)
  - Hash Join
  - Stop/PQ

```
5, 8, 13, 20, 37, ...
```

```
13, 8, 37, 5, 20, ...
```
Optimization of Stop in a Pipeline

Priority Queue

Pipelined Ops (e.g. filter, NLJ)

Scan(hotels)

<ritz, $200>
<carlton, $100>
<plaza, $259> ...

<YMCA, $25>
<Motel 5, $49>
<Days Inn, $45>

bound: $49
Push Down Stop Operators Through Pipeline Breakers

- Sometimes, pipelined plan is not attractive
- Or, pipelined plan is not possible (no indexes)
- In these cases, apply Stop as early as possible in order to reduce size of intermediate results
- Analogous to predicate push-down in traditional query optimization
Conservative Approach

- example:
  ```sql
  SELECT *
  FROM Hotels h, Cities c
  WHERE h.city = c.name
  ORDER BY price
  STOP AFTER 10;
  ```
- look at integrity constraints
- Push-down through non-reductive operators
- Every hotel qualifies join (join is non-reductive)
- **Stop** at the top necessary if a hotel matches several cities
Aggressive Approach

• Conservative approach not always applicable

• example:
  SELECT *
  FROM Hotels h, Cities c
  WHERE h.city = c.name
      AND c.state = Wisconsin
  ORDER BY price
  STOP AFTER 10;

• partition on price before join

• use DB statistics
Conservative vs. Aggressive

• If Conservative applicable, do it.
• Aggressive:
  – can reduce the cost of other operations significantly (e.g., joins, sorts)
  – (unanticipated) restarts due to poor partitioning (i.e., bad statistics) cause additional costs
• Conservative is being implemented by IBM
• No commercial product is Aggressive yet
Union Queries (Parallel System)

SELECT * 
FROM Hotels 
ORDER BY price 
STOP AFTER 10;

Client

UNION

Stop(10)

Hotels 1
Server 1

Stop(10)

Hotels 2
Server 2

Stop(10)

Hotels 3
Server 3
Top N and Semi-joins

• **Idea**
  – keep rids, project out columns at the beginning
  – at the end use rids to refetch columns

• **Tradeoff**
  – reduces cost of joins, sorts etc. because intermediate results are smaller
  – additional overhead to refetch columns

• **Attractive for top N because N limits refetch**
Skyline Queries

- Hotels which are close to the beach and cheap.

Literatur: Maximum Vector Problem. [Kung et al. 1975]
Syntax of Skyline Queries

• Additional **SKYLINE OF** clause  
  [Börszönyi, Kossmann, Stocker 2001]

• **Cheap & close to the beach**

```
SELECT *
FROM   Hotels
WHERE  city = 'Nassau'
SKYLINE OF distance MIN, price MIN;
```
Flight Reservation

• Book flight from Washington DC to San Jose

```sql
SELECT * 
FROM Flights 
WHERE depDate < 'Nov-13'
SKYLINE OF price MIN,
distance(27750, dept) MIN,
distance(94000, arr) MIN,
('Nov-13' - depDate) MIN;
```
Visualisation (VR)

• Skyline of NY (visible buildings)

SELECT * FROM Buildings
WHERE city = `New York`
SKYLINE OF h MAX, x DIFF, z MIN;
Location-based Services

- Cheap Italian restaurants that are close
- Query with current location as parameter

```
SELECT    *
FROM      Restaurants
WHERE     type = `Italian`
SKYLINE OF price MIN, d(addr, ?) MIN;
```
Skyline and Standard SQL

• Skyline can be expressed as nested Queries

```sql
SELECT * 
FROM Hotels h 
WHERE NOT EXISTS ( 
  SELECT * FROM Hotels 
  WHERE h.price >= price AND h.d >= d 
  AND (h.price > price OR h.d > d))
```

• Such queries are quite frequent in practice

• The response time is disastrous
Naive Algorithm

- Nested Loops
  - compare every point with every other point

FOR i=1 TO N
  D = FALSE;
  j = 1;
  WHILE (NOT D) AND (j <= N)
    D = dominate(a[j], a[i]);
    j++;
  END WHILE
END FOR
Block Nested-Loops Algorithm

• **Problems of naive algorithm**
  – N scans of entire table
    • (many I/Os if table does not fit in memory
  – points are compared twice

• **Block Nested Loops Algorithm**
  – keep *window* of *uncomparable points*
  – demote points not in window into temp file

• **Assessment**
  – N / windowsize scans through DB
  – no pairs of points are every compared twice
BNL Example

Input: ABCDEFG

Window size = 2

<table>
<thead>
<tr>
<th>Step</th>
<th>Window</th>
<th>Input</th>
<th>Temp</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,2</td>
<td>AB</td>
<td>CDEFG</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>AC</td>
<td>DEFG</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4-7</td>
<td>AC</td>
<td>EFG</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>EFG</td>
<td>AC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9-11</td>
<td>EG</td>
<td>AC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>ACEG</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
BNL Variants

• „Self-organizing List“
  – move hits to the beginning of window
  – saves CPU cost for comparisons

• „Replacement“
  – maximize „volume“ of window
  – additional CPU overhead
  – less iterations because more effective window
Divide & Conquer Algorithm

• [Kung et al. 1975]

• Approach:
  – Partition the table into two sets
  – apply algo recursively to both sets
  – Merge the two sets; special trick when merge

• Best algorithm in „worst case“
  \[ O(n \times (\log n)^{(d-2)}) \]

• Poor in best case (and expected case)

• Bad if DB does not fit in main memory
Variants of D&C Algos

• **M-way Partitioning**
  – Partition into M sets (rather than 2)
    • choose M so that results fit in main memory
  – Extend Merge Algorithm to M-way merge
  – Optimize „Merge Tree“
  – Much better I/O behavior

• **Early Skyline**
  – Eliminate points „on-the-fly“
  – saves both IO and CPU cost
1. Sort points according to “x, y”
2. Compare point only with previous point
Online Algorithms

• **Return first results immediately**
  – Give response time guarantees for first $x$ points

• **Incremental Evaluation**
  – get better „big picture“ the longer the algo runs
  – generate full Skyline if runs long enough

• **Fairness; User controls where to invest**

• **Correct** – never return non-Skyline points

• **General, can be integrated well into DBMS**
Online Skyline Algorithmus

[Kossmann, Ramsak, Rost 2002]

- **Divide & Conquer Algorithmus**
  - *Look for Nearest Neighbor* (e.g., using R* Baum)
  - *Partition space into Bounding Boxes*
  - *Look for Nearest Neighbors in Bounding Boxes*
Online Skyline Algorithmus

[Kossmann, Ramsak, Rost 2002]

• Divide & Conquer Algorithmus
  – Look for Nearest Neighbor (e.g., using R* Baum)
  – Partition space into Bounding Boxes
  – Look for Nearest Neighbors in Bounding Boxes

• Correctness - 2 Lemmas
  – Every Nearest Neighbor is a Skyline point
  – Every Nearest Neighbor in a Bounding Box is a Skyline point
Der NN Algorithmus
Der NN Algorithmus
Der NN Algorithmus
Der NN Algorithmus
Der NN Algorithmus
Der NN Algorithmus
Implementierung

• **NN Search with R* Tree, UB Tree, ...**
  – Bounding Boxes easy to take into account
  – Other predicates easy to take into account
  – Efficient and highly optimized in most DBMS

• **For \( d > 2 \) bounding boxes overlap**
  – need to eliminate duplicates
  – Merge Bounding Boxes
  – Propagate NNs

• **Algorithm works well for mobile applications**
  – Parameterized search in R* Tree
Experimentelle Bewertung

M-way D&C

NN (prop)

NN (hybrid)

5 dimensions, anti-correlated

Time (in sec)

NNs found
User Control
Benutzerkontrolle

User clicks here!
„distance“ more important than „price“
User Control

distance

price
User Control
Online Aggregation

• Get approximate result very quickly
• Result (conf. intervals) get better over time
• Based on random sampling (difficult!)
• No product supports this yet

SELECT cust, avg(price)
FROM Order
GROUP BY cust

<table>
<thead>
<tr>
<th>Cust</th>
<th>Avg</th>
<th>+/-</th>
<th>Conf</th>
</tr>
</thead>
<tbody>
<tr>
<td>Heinz</td>
<td>1375</td>
<td>5%</td>
<td>90%</td>
</tr>
<tr>
<td>Ute</td>
<td>2000</td>
<td>5%</td>
<td>90%</td>
</tr>
<tr>
<td>Karin</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
Time

• There are two kinds of times
  – application specific; e.g., order date, shipping date
  – system specific; when did order enter system
  – bi-temporal data model

• System time can be simulated in App
  – but cumbersome
  – most systems have built-in features for System time

• There is no update – only a new version of data
  – supports application-defined UNDO
  – (you can spend a whole lecture on this!)
Time Travel

• Give results of query AS OF a certain point in time
• Idea: Database is a sequence of states
  – DB1, DB2, DB3, … DBn
  – Each commit of a transaction creates a new state
  – To each state, associate time stamp and version number
• Idea builds on top of „serialization“
  – Time travel mostly relevant for OLTP system in order to get reproducable results or recover old data
• Implementation (Oracle - Flashback)
  – Leverage versioned data store + snapshot semantics
  – Chaining of versions of records
  – Specialized index structures (add time as a „parameter“)
Time Travel Syntax

• Give me avg(price) per customer as of last week
  
  SELECT  cust, avg(price)
  FROM     Order AS OF MAR-23-2007
  GROUP BY cust

• Can use timestamp or version number
  – Special built-in functions to convert timestamp <-> von
  – None of this is standardized (all Oracle specific)
• Now Temporal Join and Temporal Aggregation
Notification (Oracle)

• Inform me when account drops below 1000

```
SELECT *
FROM accounts a
WHEN a.balance < 1000
```

• Based on temporal model
  – Query state transitions; monitor transition: false->true
  – No notification if account stays below 1000

• Some issues:
  – How to model „delete“?
  – How to create an RSS / XML stream of events?
DBMS for Data Warehouses

• **ROLAP – Extend RDBMS**
  – Special Star-Join Techniques
  – Bitmap Indexes
  – Partition Data by Time (Bulk Delete)
  – Materialized Views

• **MOLAP – special multi-dimensional systems**
  – Implement cube as (multi-dim.) array
  – Pro: potentially fast (random access in array)
  – Problem: array is very sparse

• **Religious war (ROLAP wins in industry)**
Overview

• Motivation and Architecture

• SQL Extensions for Data Warehousing (DSS)
  – Algorithms and Query Processing Techniques

• **Column Stores, Vector Databases**

• Parallel Databases

• Operational BI
Row Store vs. Column Store

- **OLTP**: many inserts of new rows
- **OLAP**: read (few) whole columns
  - denormalization adds to this observation
Advantages of Column Stores

• **Data Locality**
  – you only read the data that you need
  – you only buffer the data that you need
  – small intermediate results ("position lists")
  – true for disk-based & in-memory systems

• **Compression**
  – lower entropy within a column than row
  – (again, important for disk-based & in-memory)

• **SIMD Instructions**
  – execute same operation on several values at once
  – (e.g., 64 bit machine with 32 bit integers -> x2)
**Query Processing in Column Stores**

```sql
SELECT sum(price) FROM Order WHERE product="ball";
```

<table>
<thead>
<tr>
<th>RowID</th>
<th>Product</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ball</td>
</tr>
<tr>
<td>2</td>
<td>net</td>
</tr>
<tr>
<td>3</td>
<td>ball</td>
</tr>
<tr>
<td>4</td>
<td>ball</td>
</tr>
<tr>
<td>5</td>
<td>racket</td>
</tr>
<tr>
<td>6</td>
<td>net</td>
</tr>
</tbody>
</table>

\[ \sigma_{\text{product="ball"}}(\text{Order}) \]

<table>
<thead>
<tr>
<th>RowID</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
</tr>
<tr>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>4</td>
<td>9</td>
</tr>
<tr>
<td>5</td>
<td>12</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
</tr>
</tbody>
</table>

\[ \Pi_{\text{sum}}(\text{sum of Price}) = 21 \]
Disadvantages of Column Stores

• Every query involves a join of the columns
  – cheap if you keep position lists sorted
  – not a problem if you always scan anyway
    • (more on that later)
• Need to “materialize” tuples; copy data
  – not a problem for aggregate queries (small results)
  – not a problem if round-trips to disk needed
  – optimizer controls best moment to “materialize”
• Every insert involves $n$ inserts (n columns)
  – that is why not good for OLTP!!!
Vectorization

• **Iterator Model (\(\rightarrow\) Bachelor courses)**
  - `open()` – `next()` – `close()` Interface of operators
  - `next()` returns (pointer to) one result tuple
  - great for composability of operators
  - great for pipelined parallelism

• **Problems of Iterator Model**
  - poor instruction cache locality
    - reload code of every operator with every tuple
  - poor use of bandwidth of “bus” (network in machine)
    - ship 32 bit pointers on 128 bit bus

• **Idea: Ship batches of tuples with every `next()` call**
  - works well in row and column store
Overview

• Motivation and Architecture

• SQL Extensions for Data Warehousing (DSS)
  – Algorithms and Query Processing Techniques

• Column Stores, Vector Databases

• **Parallel Databases**

• Operational BI
Parallel Database Systems

• Why is a query slow?
  – bottlenecks
  – it needs to do a lot of work
  – (performance bugs; e.g., wrong plan)

• How to make it fast, if it is just a lot of work?
  – partitioning and replication
  – exploit different forms of parallelism

• Reference: DeWitt, Gray: CACM 1992
Why are response times long?

• **Because operations take long**
  – cannot travel faster than light
  – delays even in „single-user“ mode
  – fix: parallelize long-running operations
    • **data partitioning for „intra-query parallelism“**

• **Because there is a bottleneck**
  – contention of concurrent requests on a resource
  – requests wait in queue before resource available
  – add resources to parallelize requests at bottleneck
    • **replication for „inter-query parallelism“**
Forms of Parallelism

• **Inter-request Parallelism**
  – several requests handled at the same time
  – principle: replicate resources
  – e.g., ATMs

• **(Independent) Intra-request Parallelism**
  – principle: divide & conquer
  – e.g., print pieces of document on several printers

• **Pipelining**
  – each „item“ is processed by several resources
  – process „items“ at different resources in parallel
  – can lead to both inter- & intra-request parallelism
Inter-request Parallelism

req 1

resp 1

resp 2

resp 3
Pipelining (Intra-request)

Example: Dish Washing
Speed-up

• Metric for intra-request parallelization

• Goal: reduce response time
  – measure response time with 1 resource
  – measure response time with N resources
  – SpeedUp(N) = RT(1) / RT(N)

• Ideal
  – SpeedUp(N) is a linear function
  – can you imagine super-linear speed-ups?
Scale-up

• **Goal**: Scales with size of the problem
  – measure response time with 1 server, unit problem
  – measure response time with N servers, N units problem
  – \( \text{ScaleUp}(N) = \frac{RT(1)}{RT(N)} \)

• **Ideal**
  – ScaleUp(N) is a constant function (1)
  – Can you imagine super scale-up?
Scale Out (transactional scale-up)

• Goal: Scale with users / jobs / transactions
  – measure throughput: 1 server, k users
  – measure throughput: N servers, k*N users
  – $\text{ScaleOut}(N) = \frac{T\text{put}(1)}{T\text{put}(N)}$

• Ideal
  – Scale-out should behave like Scale-Up
  – (often terms are used interchangeably; but worth-while to notice the differences)

• Scale-out and down in Cloud Computing
  – the ability of a system to adapt to changes in load
  – often measured in $\ (or at least involving cost)$
Why is speed-up sub-linear?
Why is speed-up sub-linear?

• **Cost for „split“ and „merge“ operation (Amdahl)**
  – those can be expensive operations
  – try to parallelize them, too

• **Interference: servers need to synchronize**
  – e.g., CPUs access data from same disk at same time
  – shared-nothing architecture

• **Skew: work not „split“ into equal-sized chunks**
  – e.g., some pieces much bigger than others
  – keep statistics and plan better
How to split a problem?

• Cost model to split problem into „p“ pieces
  \[
  \text{Cost}(p) = a * p + \frac{(b * K)}{p}
  \]
  – a: constant overhead per piece for split & merge
  – b: constant overhead per item of the problem
  – K: total number of items in the problem
  – cost for split and data processing may differ!!!

• Minimize this function
  – simple calculus: \( \text{Cost}(p)' = 0 \); \( \text{Cost}(p)'' > 0 \)
    \[
    p = \sqrt{\frac{b * K}{a}}
    \]

• Do math if you can!!!
Distributed & Parallel Databases

• Distributed Databases (e.g., banks)
  – partition the data
  – install database nodes at different locations
  – keep partitions at locations where frequently needed
  – if beneficial replicate partitions / cache data
  – goal: reduce communication cost

• Parallel Databases (e.g., Google)
  – partition the data
  – install database nodes within tightly-coupled network
  – goal: speed-up by parallel queries on partitions
Kinds of Parallel Databases

• **Shared Nothing**
  – each node has its own disk, main memory, CPU
  – nodes communicate via message passing

• **Shared Disk**
  – data is stored persistently on disk accessible by all
  – nodes fetch data from (shared) disk as needed

• **Shared Memory**
  – a node has a CPU (+ cache)
  – nodes communicate via shared memory
Scans in Shared Nothing

- `SELECT * FROM Emp WHERE salary > 1000;`

Diagram:
- **σ**
  - (Helga, 2000)
  - (Hubert, 150)
  - ...

- **∪**
  - (Peter, 20)
  - (Rhadia, 15000)
  - ...

- **σ**
Scans in Shared Nothing

• **Approach**
  – each node has a (horizontal) partition of DB
  – each node carries out scan + filter locally
  – each node sends results to dedicated node
  – dedicated node carries out U for final result

• **Assessment**
  – scales almost linearly
  – skew in communicating results may limit scalability
Joins in Shared Nothing (V1)

• **Approach**
  – Table 1 is horizontally partitioned across nodes
  – ship (entire) Table 2 to all nodes
  – carry out $P_i(T1) \bowtie T2$ at each node
  – compute $U$ of all local joins

• **Assessment**
  – scales well if there is an efficient broadcast
  – even better if Table 2 is already replicated everywhere
    • or if the database is shared (see later)
Joins in Shared Nothing (V2)

• **Approach**
  – partition Table 1 using Function h
    • ship partitions to different nodes accordingly
  – partition Table 2 using Function h
    • ship partitions to different nodes accordingly
  – carry out local joins at each node
  – compute U of all local joins

• **Assessment**
  – ships both Tables entirely through network
  – sensitive to skew during partitioning
    • can be fixed by building histograms in a separate phase
  – computationally as good as hash join
Encapsulating Parallelism

[Graefe, 1992]
Encapsulating Parallelism (Plans)

```
SELECT x, y, z
FROM T1, T2, T3
WHERE T1.a = T2.b AND T2.b = T3.c;
```
Joins in Shared Memory

• Approach
  – build hash table of Table 2 in shared memory
  – parallel probe hash table with Table 1

• Assessment
  – resource contention on bus during probe
  – build phase cannot be parallelized
  – (rarely a good idea; need special HW)
Why are PDDBs so cool? ;-)

- **Data is a „resource“ (just like a server)**
  - data can be a bottleneck if it is updated
  - data can be replicated in order to improve throughput

- **Data is a „problem“**
  - data can be partitioned in good and poor ways
  - partitioning can be done statically and dynamically
  - if statically, then „split“ operation is free

- **Data can be used for scalability experiments**
  - you can nicely show all
How to partition data?

• (here: horizontal partitioning only)
• **Step 1:** Need to determine partitioning factor
  – very difficult task; depends on many factors
• **Step 2:** Determine partitioning method
  – Round-robin: good for load balancing
  – Predicate-based: good for certain queries (e.g., sort)
  – Hashing: good for “key” look-ups and updates
  – **Sharding:** partition dependent tables in the same way
• **Step 3:** Determine allocation
  – which partition to replicate and how often
  – where to store replicas of each partition
Response Time Cost Models

• Estimate the response time of a query plan
  – Consider independent parallelism
    • max
  – Consider pipelined parallelism
    • materialized front + max
  – Consider resource contention
    • consumption vector + max

• [Ganguly et al., 1992]
Independent Parallelism

• Response Time = $\max(\text{RT(join1)}, \text{RT(join2)})$
  – assuming nothing else is happening
Pipelined Parallelism

\[ \text{max}(\text{RT}(\text{join2}), \text{RT}(\text{build1})) + \text{max}(\text{RT}(\text{probe1}), \text{RT}(\text{probe3})) \]
Resource Contention

- What if join1, join3 executed on same node?
- Model resource consumption as vector
  - Consumption(probe3) = (m1, m2, m3, network)
- Add resource consumption of parallel operators
  - E.g., Consumption(probe3) + Consumption(probe1)
- Model capacity as capacity vector
  - Capacity = (m1, m2, m3, network)
- Match aggregated consumption with capacity
  - May result in higher response times
Summary

• Improve Response Times by „partitioning“
  – divide & conquer approach
  – works extremely well for databases and SQL
  – do the math for query optimization

• Improve Throughput by „inter-query“ parallelism
  – limited in SQL because of concurrency control

• Parallelism problems in databases
  – resource contention (e.g., lock conflicts, network)
  – skew and poor load balancing

• Special kinds of experiments for scalability
  – speed-up and scale-up experiments
Overview

• Motivation and Architecture

• SQL Extensions for Data Warehousing (DSS)
  – Algorithms and Query Processing Techniques

• Column Stores, Vector Databases

• Parallel Databases

• Operational BI
Operational BI

• Sometimes you need fresh data for decisions
  – you need to be transactionally consistent
  – or you cannot afford delay of ETL

• Examples
  – passenger lists at airlines
  – route visitors at Disney resorts
  – ...

Amadeus Workload

• **Passenger Booking Database**
  – ~ 600 GB of raw data (two years of bookings)
  – single denormalized table (for now)
  – ~ 50 attributes: flight-no, name, date, ..., many flags

• **Query Workload**
  – up to 4000 queries / second
  – latency guarantees: 2 seconds
  – today: only pre-canned queries allowed

• **Update Workload**
  – avg. 600 updates per second (1 update per GB per sec)
  – peak of 12000 updates per second
  – data freshness guarantee: 2 seconds
Amadeus Query Examples

• Simple Queries
  – Print passenger list of Flight LH 4711
  – Give me Hon Circle members booked Zurich to Boston

• Complex Queries
  – Give me all Heathrow passengers that need special assistance (e.g., after terror warning)

• Problems with State-of-the Art
  – Simple queries work only because of mat. views
    • multi-month project to implement new query / process
  – Complex queries do not work at all
Goals

• **Predictable (= constant) Performance**
  – independent of updates, query types, ...

• **Meet SLAs**
  – latency, data freshness

• **Affordable Cost**
  – ~ 1000 commodity servers are okay
    – (compare to mainframe)

• **Meet Consistency Requirements**
  – monotonic reads and writes (ACID not needed)

• **Respect Hardware Trends**
  – main-memory, NUMA, large data centers

• **Allow any kind of ad-hoc query** (e.g., terror, volcano)
New Approaches for Operational BI

• Have all data in one database!

• Use a traditional DBMS with Snapshot Isolation
  – SI addresses lock conflicts between OLAP + OLTP

• Delta Indexing (+ SI)
  – read vs. write optimized data structures

• Crazy new ideas
  – e.g. Crescando and Swissbox
Snapshot Isolation

- When a TA starts it receives a timestamp, T.
- All reads are carried out *as of* the DB version of T.
  - Need to keep historic versions of all objects!!!
- All writes are carried out in a separate buffer.
  - Writes only become visible after a commit.
- When TA commits, DBMS checks for conflicts
  - Abort TA1 with timestamp T1 if exists TA2 such that
    - TA2 committed after T1 and before TA1
    - TA1 and TA2 updated the same object

**Snapshot Isolation and serializability?** [Berenson+95]

**Advantages/disadv. of Snapshot Isolation?**
### SI and Lost Update

<table>
<thead>
<tr>
<th>Step</th>
<th>$T_1$</th>
<th>$T_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>BOT</td>
<td></td>
</tr>
<tr>
<td>2.</td>
<td>read($A$)</td>
<td></td>
</tr>
<tr>
<td>3.</td>
<td></td>
<td>BOT</td>
</tr>
<tr>
<td>4.</td>
<td></td>
<td>read($A$)</td>
</tr>
<tr>
<td>5.</td>
<td></td>
<td>write($A$)</td>
</tr>
<tr>
<td>6.</td>
<td></td>
<td>commit</td>
</tr>
<tr>
<td>7.</td>
<td>write($A$)</td>
<td></td>
</tr>
<tr>
<td>8.</td>
<td>commit</td>
<td></td>
</tr>
</tbody>
</table>
### SI and Lost Update (ctd.)

<table>
<thead>
<tr>
<th>Step</th>
<th>$T_1$</th>
<th>$T_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td></td>
<td>BOT</td>
</tr>
<tr>
<td>2.</td>
<td></td>
<td>read(A)</td>
</tr>
<tr>
<td>3.</td>
<td></td>
<td>write(A)</td>
</tr>
<tr>
<td>4.</td>
<td>BOT</td>
<td></td>
</tr>
<tr>
<td>5.</td>
<td>read(A)</td>
<td></td>
</tr>
<tr>
<td>6.</td>
<td>write(A)</td>
<td>commit</td>
</tr>
<tr>
<td>7.</td>
<td>commit</td>
<td></td>
</tr>
<tr>
<td>8.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

SI reorders R1(A) and W2(A) -> not serializ. -> abort of T1
## SI and Uncommitted Read

<table>
<thead>
<tr>
<th>Step</th>
<th>$T_1$</th>
<th>$T_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>BOT</td>
<td></td>
</tr>
<tr>
<td>2.</td>
<td>read(A)</td>
<td></td>
</tr>
<tr>
<td>3.</td>
<td>write(A)</td>
<td></td>
</tr>
<tr>
<td>4.</td>
<td></td>
<td>BOT</td>
</tr>
<tr>
<td>5.</td>
<td></td>
<td>read(A)</td>
</tr>
<tr>
<td>6.</td>
<td></td>
<td>write(A)</td>
</tr>
<tr>
<td>7.</td>
<td>read(B)</td>
<td>...</td>
</tr>
<tr>
<td>8.</td>
<td>abort</td>
<td></td>
</tr>
</tbody>
</table>
Discussion: Snapshot Isolation

- **Concurrency and Availability**
  - No read or write of a TA is ever blocked
  - (Blocking only happens when a TA commits.)

- **Performance**
  - Need to keep write-set of a TA only
  - Very efficient way to implement aborts
  - Often keeping all versions of an object useful anyway
  - No deadlocks, but unnecessary rollbacks
  - Need not worry about phantoms (complicated with 2PL)

- **Correctness (Serializability): Write Skew**
  - Checking integrity constraint also happens in the snapshot
  - Two concurrent TAs update different objects
  - Each update okay, but combination not okay
  - Example: Both doctors sign out...
### Example: One doctor on duty!

<table>
<thead>
<tr>
<th>Step</th>
<th>$T_1$</th>
<th>$T_2$</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>BOT (A, duty); (B, duty)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.</td>
<td>write(A, free)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4.</td>
<td></td>
<td>BOT write(B, free)</td>
<td></td>
</tr>
<tr>
<td>5.</td>
<td>check-constraint</td>
<td></td>
<td>Okay: (B, duty)</td>
</tr>
<tr>
<td>6.</td>
<td></td>
<td>check-constraint</td>
<td>Okay: (A, duty)</td>
</tr>
<tr>
<td>7.</td>
<td>commit</td>
<td>commit</td>
<td></td>
</tr>
<tr>
<td>8.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9.</td>
<td></td>
<td></td>
<td>Constraint violated!!!</td>
</tr>
</tbody>
</table>

N.B. Example can be solved if check part of DB commit. Impossible to solve at the app level.
New Approaches for Operational BI

• Have all data in one database!

• Use a traditional DBMS with Snapshot Isolation
  – SI addresses lock conflicts between OLAP + OLTP

• **Delta Indexing (+ SI)**
  – read vs. write optimized data structures

• Crazy new ideas
  – e.g. Crescando and Swissbox
Delta Indexing

- **Key Idea** (e.g., SAP Hana)
  - have a write optimized data structure (called $\Delta$)
  - have a read optimized data structure (called “main”)
  - all updates create $\Delta$ records in $\Delta$
  - all queries need to be executed against $\Delta$ and main
  - periodically merge $\Delta$ and main so that $\Delta$ stays small

- **Assessment**
  - balance read and write performance,
    - a number of low-level optimizations possible
  - SI can nicely be integrated, allows relaxed consistency
    - e.g. Movies (Blunschi et al.)
  - efficient merge: sort and rebuild
    - but merge is potential bottleneck
Delta Indexing

put(k, value) → get(k, version)
New Approaches for Operational BI

• Have all data in one database!

• Use a traditional DBMS with Snapshot Isolation
  – SI addresses lock conflicts between OLAP + OLTP

• Delta Indexing (+ SI)
  – read vs. write optimized data structures

• Crazy new ideas
  – e.g. Crescando and Swissbox
What is Crescando?

• A distributed (relational) table: MM on NUMA
  – horizontally partitioned
  – distributed within and across machines
• Query / update interface
  – SELECT * FROM table WHERE <any predicate>
  – UPDATE table SET <anything> WHERE <any predicate>
• Some nice properties
  – constant / predictable latency & data freshness
  – solves the Amadeus use case
  – support for Snapshot Isolation, monotonic writes
Design

• Operate MM like disk in shared-nothing architect.
  – **Core ~ Spindle** (many cores per machine & data center)
  – all data kept in main memory (log to disk for recovery)
  – each core scans one partition of data all the time

• Batch queries and updates: shared scans
  – do trivial MQO (at scan level on system with single table)
  – control read/update pattern -> no data contention

• Index queries / not data
  – just as in the stream processing world
  – predictable+optimizable: rebuild indexes every second

• Updates are processed before reads
Crescendo on 1 Machine (N Cores)
Crescando on 1 Core
Scanning a Partition
Scanning a Partition

- Record 0
- Snapshot n
- Snapshot n+1
- Read Cursor
- Write Cursor

Merge cursors
Scanning a Partition

Record 0

Read Cursor

Write Cursor

Snapshot n

Snapshot n+1

Build indexes for next batch of queries and updates

Merge cursors

Scanning a Partition
Crescando @ Amadeus

Queries (Oper. BI) → Aggregator → Key / Value → Mainframe → Update stream (queue) → Crescando Nodes

Transactions (OLTP)
Implementation Details

• **Optimization**
  – decide for batch of queries which indexes to build
  – runs once every second (must be fast)

• **Query + update indexes**
  – different indexes for different kinds of predicates
  – e.g., hash tables, R-trees, tries, ...
  – must fit in L2 cache (better L1 cache)

• **Probe indexes**
  – Updates in right order, queries in any order

• **Persistence & Recovery**
  – Log updates / inserts to disk (not a bottleneck)
What is SharedDB?

• Implementation of relational algebra
  – Joins, Group-Bys, Sorting, ...

• Massive sharing of operators of the same kind
  – Joins with the same join predicate
  – Sorts with the same sorting key

• Natural extension of key Crescando idea
  – Apply operator on UNION of data of many queries
  – Route the results to the right client

• Complements nicely with Crescando
  – Crescando: storage layer with predicate push-down
  – SharedDB: query processor
**Set of Queries**

- **Q1**
  
  ```
  SELECT *  
  FROM R,S  
  WHERE R.id = S.id  
  AND R.city = ?  
  AND S.date = ?
  ```

- **Q2**
  
  ```
  SELECT *  
  FROM R,S  
  WHERE R.id = S.id  
  AND R.name = ?  
  AND S.date < ?
  ```

- **Q3**
  
  ```
  SELECT *  
  FROM R,S  
  WHERE R.id = S.id  
  AND R.address = ?  
  AND S.date > ?
  ```

**Traditional Query Processing**

- **Q1**
  
  - R
  - S
  - Q1

- **Q2**
  
  - R
  - S
  - Q2

- **Q3**
  
  - R
  - S
  - Q3

**Shared Query Processing**

- **Q1, Q2, Q3**
  
  - R
  - S
  - U
  - Q1, Q2, Q3

  ```
  \Gamma_{query_id}
  ```

  ```
  R.id = S.id &&
  R.query_id = \Gamma_{query_id}
  ```
Set of Active Queries

Q1  Q2  Q3

Shared Group By

Shared Join

Shared Sort

Shared Duplicate Elimination (Union)

Crescando Storage Engine

Shared Query Processing Engine

Set of Active Queries
Global / Always-on Query Plan
Load Interaction between Light and Heavy Queries

Throughput (Web Interactions/second)

Percentage of Heavy Queries in the Workload
Overview of Components
Take Home Messages

• Big Data (Data-driven Intelligence) is not new
  – 40 years of experience in database technology
  – “volume” pretty much under control, unbeatable perf. (!)
  – “complexity” addressed with SQL extensions
  – many success stories

• What are the short-comings of data warehouses?
  – “diversity” – only 20% of data is relational
    • very expensive to squeeze other 80% into tables
  – “fast” – ETL is cumbersome and painful
    • in-situ processing of data much better
  – “complexity” – at some point, SQL hits its limits
    • success kills (-> similar story with Java)

• Bottom line: Flexibility (time to market) vs. Cost