

Miscellaneous Topics



Amol Deshpande
CMSC424

Topics

- Distributed Databases and Transactions
- Cloud Computing
 - ★ Data centers, Map-reduce, NoSQL Systems
- OLAP/Data Warehouses
- Object Oriented, Object Relational
- Information Retrieval



Distributed Database System

- A distributed database system consists of loosely coupled sites that share no physical component
- Database systems that run on each site are independent of each other
- Transactions may access data at one or more sites



Homogeneous Distributed Databases

- In a homogeneous distributed database
 - All sites have identical software
 - Are aware of each other and agree to cooperate in processing user requests.
 - Each site surrenders part of its autonomy in terms of right to change schemas or software
 - Appears to user as a single system
- In a heterogeneous distributed database
 - Different sites may use different schemas and software
 - ▶ Difference in schema is a major problem for query processing
 - ▶ Difference in software is a major problem for transaction processing
 - Sites may not be aware of each other and may provide only limited facilities for cooperation in transaction processing



Data Replication

- A relation or fragment of a relation is **replicated** if it is stored redundantly in two or more sites.
- **Full replication** of a relation is the case where the relation is stored at all sites.
- Fully redundant databases are those in which every site contains a copy of the entire database.



Data Replication (Cont.)

■ Advantages of Replication

- **Availability**: failure of site containing relation r does not result in unavailability of r if replicas exist.
- **Parallelism**: queries on r may be processed by several nodes in parallel.
- **Reduced data transfer**: relation r is available locally at each site containing a replica of r .

■ Disadvantages of Replication

- Increased cost of updates: each replica of relation r must be updated.
- Increased complexity of concurrency control: concurrent updates to distinct replicas may lead to inconsistent data unless special concurrency control mechanisms are implemented.
 - ▶ One solution: choose one copy as **primary copy** and apply concurrency control operations on primary copy



Data Fragmentation

- Division of relation r into fragments r_1, r_2, \dots, r_n which contain sufficient information to reconstruct relation r .
- **Horizontal fragmentation**: each tuple of r is assigned to one or more fragments
- **Vertical fragmentation**: the schema for relation r is split into several smaller schemas
 - All schemas must contain a common candidate key (or superkey) to ensure lossless join property.
 - A special attribute, the tuple-id attribute may be added to each schema to serve as a candidate key.



Horizontal Fragmentation of *account* Relation

<i>branch_name</i>	<i>account_number</i>	<i>balance</i>
Hillside	A-305	500
Hillside	A-226	336
Hillside	A-155	62

$$account_1 = \sigma_{branch_name="Hillside"}(account)$$

<i>branch_name</i>	<i>account_number</i>	<i>balance</i>
Valleyview	A-177	205
Valleyview	A-402	10000
Valleyview	A-408	1123
Valleyview	A-639	750

$$account_2 = \sigma_{branch_name="Valleyview"}(account)$$



Vertical Fragmentation of *employee_info* Relation

<i>branch_name</i>	<i>customer_name</i>	<i>tuple_id</i>
Hillside	Lowman	1
Hillside	Camp	2
Valleyview	Camp	3
Valleyview	Kahn	4
Hillside	Kahn	5
Valleyview	Kahn	6
Valleyview	Green	7

$deposit_1 = \Pi_{branch_name, customer_name, tuple_id}(employee_info)$

<i>account_number</i>	<i>balance</i>	<i>tuple_id</i>
A-305	500	1
A-226	336	2
A-177	205	3
A-402	10000	4
A-155	62	5
A-408	1123	6
A-639	750	7

$deposit_2 = \Pi_{account_number, balance, tuple_id}(employee_info)$



Advantages of Fragmentation

- Horizontal:
 - allows parallel processing on fragments of a relation
 - allows a relation to be split so that tuples are located where they are most frequently accessed
- Vertical:
 - allows tuples to be split so that each part of the tuple is stored where it is most frequently accessed
 - tuple-id attribute allows efficient joining of vertical fragments
 - allows parallel processing on a relation
- Vertical and horizontal fragmentation can be mixed.
 - Fragments may be successively fragmented to an arbitrary depth.



Data Transparency

- **Data transparency:** Degree to which system user may remain unaware of the details of how and where the data items are stored in a distributed system
- Consider transparency issues in relation to:
 - Fragmentation transparency
 - Replication transparency
 - Location transparency



Naming of Data Items - Criteria

1. Every data item must have a system-wide unique name.
2. It should be possible to find the location of data items efficiently.
3. It should be possible to change the location of data items transparently.
4. Each site should be able to create new data items autonomously.

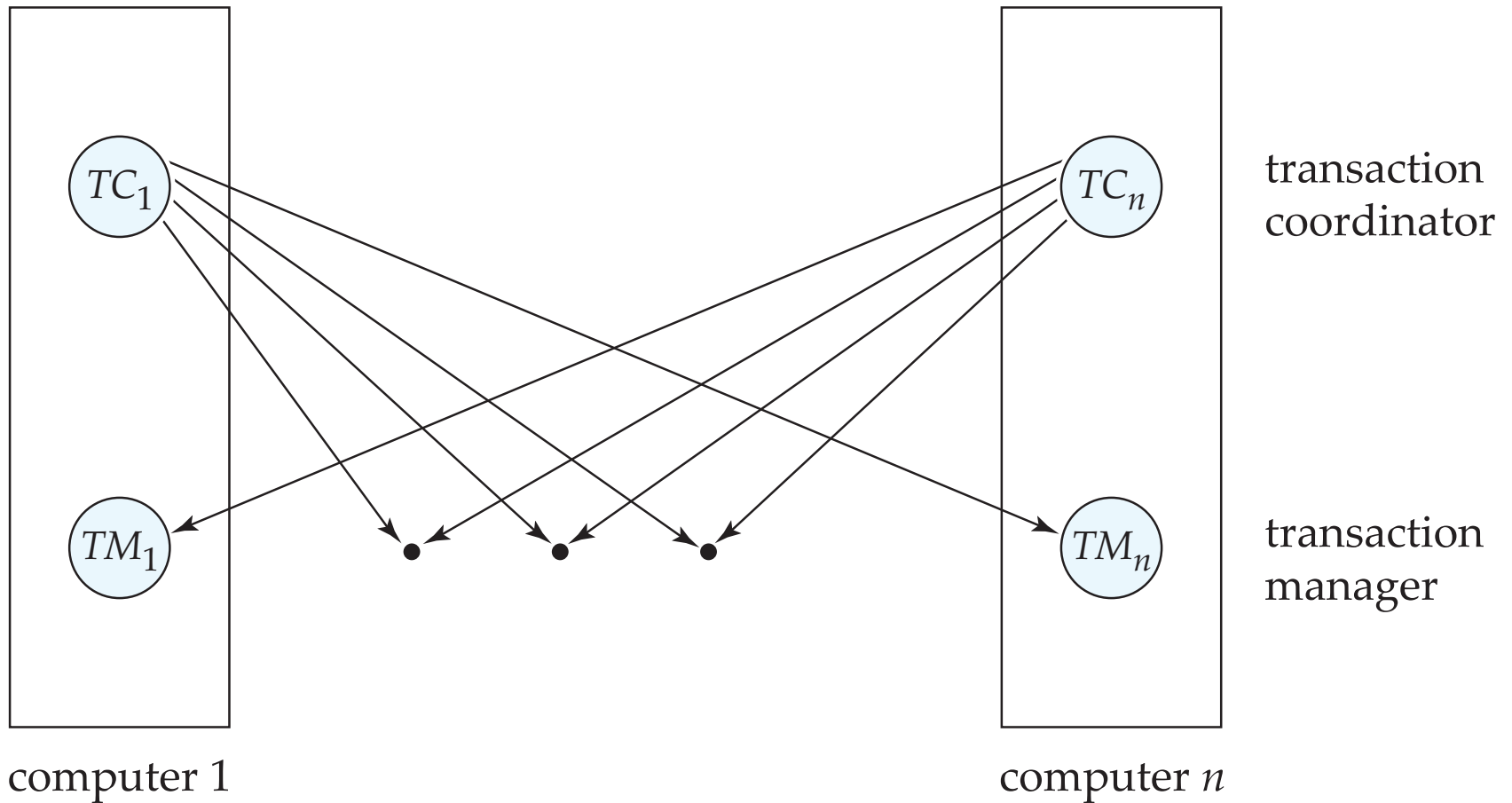


Distributed Transactions

- Transaction may access data at several sites.
- Each site has a local **transaction manager** responsible for:
 - Maintaining a log for recovery purposes
 - Participating in coordinating the concurrent execution of the transactions executing at that site.
- Each site has a **transaction coordinator**, which is responsible for:
 - Starting the execution of transactions that originate at the site.
 - Distributing subtransactions at appropriate sites for execution.
 - Coordinating the termination of each transaction that originates at the site, which may result in the transaction being committed at all sites or aborted at all sites.



Transaction System Architecture





System Failure Modes

- Failures unique to distributed systems:
 - Failure of a site.
 - Loss of messages
 - ▶ Handled by network transmission control protocols such as TCP-IP
 - Failure of a communication link
 - ▶ Handled by network protocols, by routing messages via alternative links
 - **Network partition**
 - ▶ A network is said to be **partitioned** when it has been split into two or more subsystems that lack any connection between them
 - Note: a subsystem may consist of a single node
- Network partitioning and site failures are generally indistinguishable.



Commit Protocols

- Commit protocols are used to ensure atomicity across sites
 - a transaction which executes at multiple sites must either be committed at all the sites, or aborted at all the sites.
 - not acceptable to have a transaction committed at one site and aborted at another
- The *two-phase commit* (2PC) protocol is widely used
- The *three-phase commit* (3PC) protocol is more complicated and more expensive, but avoids some drawbacks of two-phase commit protocol. This protocol is not used in practice.



Two Phase Commit Protocol (2PC)

- Assumes **fail-stop** model – failed sites simply stop working, and do not cause any other harm, such as sending incorrect messages to other sites.
- Execution of the protocol is initiated by the coordinator after the last step of the transaction has been reached.
- The protocol involves all the local sites at which the transaction executed
- Let T be a transaction initiated at site S_i , and let the transaction coordinator at S_i be C_i



Phase 1: Obtaining a Decision

- Coordinator asks all participants to *prepare* to commit transaction T_i .
 - C_i adds the records **<prepare T >** to the log and forces log to stable storage
 - sends **prepare T** messages to all sites at which T executed
- Upon receiving message, transaction manager at site determines if it can commit the transaction
 - if not, add a record **<no T >** to the log and send **abort T** message to C_i
 - if the transaction can be committed, then:
 - add the record **<ready T >** to the log
 - force *all records* for T to stable storage
 - send **ready T** message to C_i



Phase 2: Recording the Decision

- T can be committed if C_i received a **ready** T message from all the participating sites: otherwise T must be aborted.
- Coordinator adds a decision record, **<commit T >** or **<abort T >**, to the log and forces record onto stable storage. Once the record stable storage it is irrevocable (even if failures occur)
- Coordinator sends a message to each participant informing it of the decision (commit or abort)
- Participants take appropriate action locally.



Handling of Failures - Site Failure

When site S_i recovers, it examines its log to determine the fate of transactions active at the time of the failure.

- Log contain **<commit T >** record: txn had completed, nothing to be done
- Log contains **<abort T >** record: txn had completed, nothing to be done
- Log contains **<ready T >** record: site must consult C_i to determine the fate of T .
 - If T committed, **redo** (T); write **<commit T >** record
 - If T aborted, **undo** (T)
- The log contains no log records concerning T :
 - Implies that S_k failed before responding to the **prepare** T message from C_i
 - since the failure of S_k precludes the sending of such a response, coordinator C_i must abort T
 - S_k must execute **undo** (T)



Handling of Failures- Coordinator Failure

- If coordinator fails while the commit protocol for T is executing then participating sites must decide on T 's fate:
 1. If an active site contains a **<commit T >** record in its log, then T must be committed.
 2. If an active site contains an **<abort T >** record in its log, then T must be aborted.
 3. If some active participating site does not contain a **<ready T >** record in its log, then the failed coordinator C_i cannot have decided to commit T .
 - Can therefore abort T ; however, such a site must reject any subsequent **<prepare T >** message from C_i
 4. If none of the above cases holds, then all active sites must have a **<ready T >** record in their logs, but no additional control records (such as **<abort T >** or **<commit T >**).
 - In this case active sites must wait for C_i to recover, to find decision.
- **Blocking problem:** active sites may have to wait for failed coordinator to recover.



Handling of Failures - Network Partition

- If the coordinator and all its participants remain in one partition, the failure has no effect on the commit protocol.
- If the coordinator and its participants belong to several partitions:
 - Sites that are not in the partition containing the coordinator think the coordinator has failed, and execute the protocol to deal with failure of the coordinator.
 - ▶ No harm results, but sites may still have to wait for decision from coordinator.
- The coordinator and the sites are in the same partition as the coordinator think that the sites in the other partition have failed, and follow the usual commit protocol.
 - ▶ Again, no harm results



Topics

- Object Oriented, Object Relational
- Client-server, Parallel, Distributed Systems
- OLAP/Data Warehouses
- Information Retrieval
- Cloud Computing
 - Data centers, Map-reduce, NoSQL Systems



Cloud Computing: Outline

- Technologies behind cloud computing
 - Data centers
 - Virtualization
 - Programming Framework: Map-reduce
 - Distributed Key-Value Stores
- Some observations about the marketplace



Cloud Computing

- Computing as a “service” rather than a “product”
 - Everything happens in the “cloud”: both storage and computing
 - Personal devices (laptops/tablets) simply interact with the cloud
- Advantages
 - Device agnostic – can seamlessly move from one device to other
 - Efficiency/scalability: programming frameworks allow easy scalability (relatively speaking)
 - Reliability
 - Cost: “pay as you go” allows renting computing resources as needed – much cheaper than building your own systems



Cloud Computing

- Basic ideas have been around for a long time (going back to 1960's)
 - Mainframes + thin clients (more by necessity)
 - Grid computing a few year ago
 - Peer-to-peer
 - Client-server models
 - ...
- But it finally works as we wished for...
 - Why now?... A convergence of several key pieces over the last few years
 - Does it really? ... Still many growing pains



Data Centers

- The key infrastructure piece that enables CC
- Everyone is building them
- Huge amount of work on deciding how to build/design them

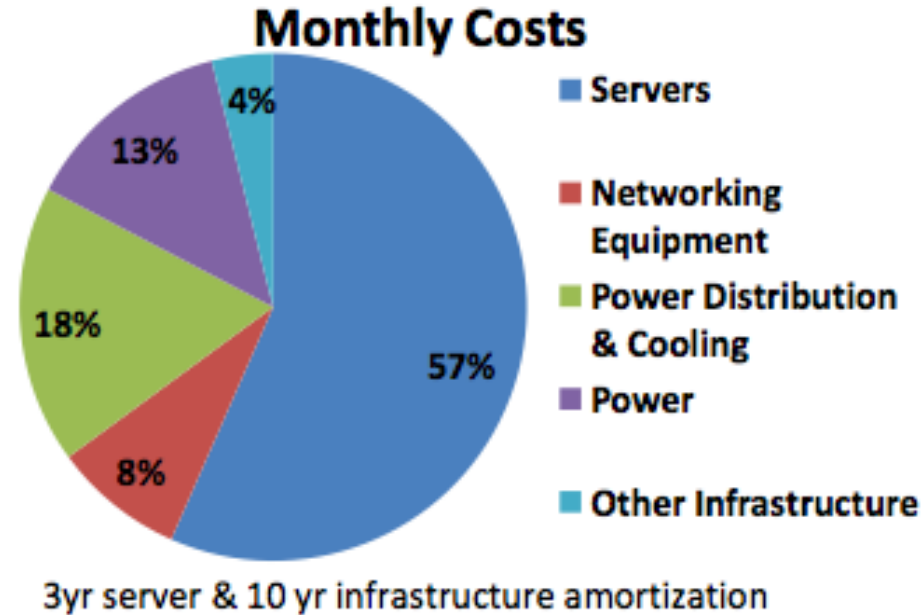




Data Centers

■ Amazon data centers: Some recent data

- 8 MW data center can include about 46,000 servers
- Costs about \$88 million to build (just the facility)
- Power a pretty large portion, but server costs still dominate



“Every day, Amazon Web Services adds enough new capacity to support all of Amazon.com’s global infrastructure through the company’s first 5 years, when it was a \$2.76B annual revenue enterprise”



Data Centers

- Power distribution
 - Almost 11% lost in distribution – starts mattering when the total power consumption is in millions
- Modular and pre-fab designs
 - Fast and economic deployments, built in a factory



Microsoft ITPAC



Amazon Perdig





Data Centers

■ Networking equipment

- Very very expensive
- Bottleneck – forces workload placement restrictions

■ Cooling/temperature/energy issues

- Appropriate placement of vents, inlets etc. a key issue
 - ▶ Thermal hotspots often appear and need to be worked around
- Overall cost of cooling is quite high
 - ▶ So is the cost of running the computing equipment
 - Both have led to issues in energy-efficient computing
- Hard to optimize PUE (Power Usage Effectiveness) in small data centers
 - ▶ ➔ may lead to very large data centers in near future



Virtualization

- Virtual machines (e.g., running Windows inside a Mac) etc. has been around for a long time
 - Used to be very slow...
 - Only recently became efficient enough to make it a key for CC
- Basic idea: run virtual machines on your servers and sell time on them
 - That's how Amazon EC2 runs
- Many advantages:
 - Security: virtual machines serves as almost impenetrable boundary
 - Multi-tenancy: can have multiple VMs on the same server
 - Efficiency: replace many underpowered machines with a few high-power machines



Virtualization

- Consumer VM products include VMWare, Parallels (for Mac) etc...
- Amazon uses “Xen” running on Redhat machines (may be old information)
 - They support both Windows and Linux Virtual Machines
- Some tricky things to keep in mind:
 - Harder to reason about performance (if you care)
 - Identical VMs may deliver somewhat different performance
- Much continuing work on the virtualization technology itself



Programming Frameworks

- Third key piece emerged from efforts to “scale out”
 - i.e., distribute work over large numbers of machines (1000's of machines)
- Parallelism has been around for a long time
 - Both in a single machine, and as a cluster of computers
- But always been considered very hard to program, especially the distributed kind
 - Too many things to keep track of
 - ▶ How to parallelize, how to distribute the data, how to handle failures etc etc..
- Google developed MapReduce and BigTable frameworks, and ushered in a new era



Programming Frameworks

- Note the difference between “scale up” and “scale out”
 - scale up usually refers to using a larger machine – easier to do
 - scale out refers to distributing over a large number of machines

- Even with VMs, I still need to know how to distribute work across multiple VMs
 - Amazon’s largest single instance may not be enough

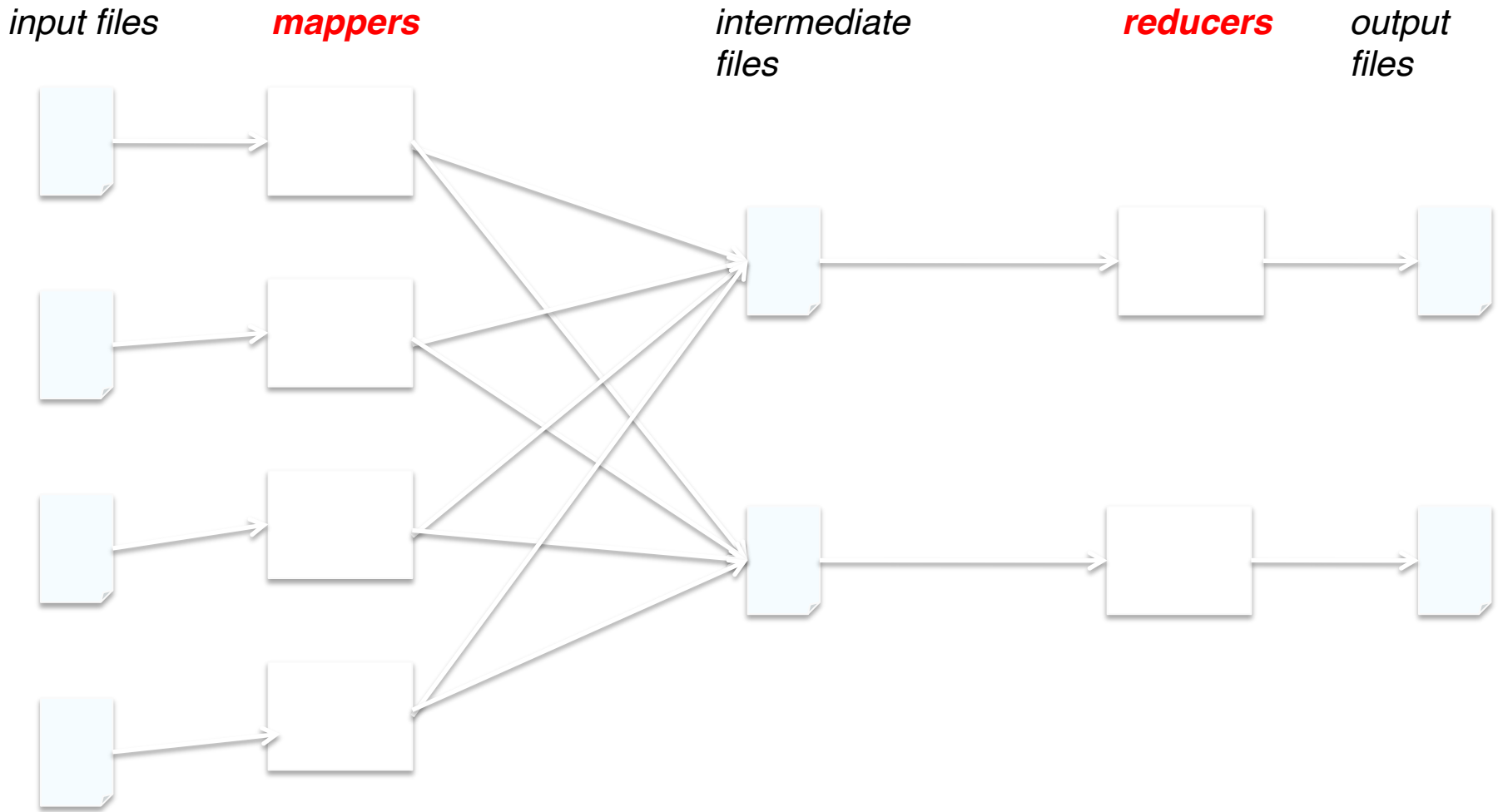


MapReduce Framework

- Provides a fairly restricted, but still powerful abstraction for programming
- Programmers write a pipeline of functions, called *map* or *reduce*
 - map programs
 - ▶ inputs: a list of “records” (record defined arbitrarily – could be images, genomes etc...)
 - ▶ output: for each record, produce a set of “(key, value)” pairs
 - reduce programs
 - ▶ input: a list of “(key, {values})” grouped together from the mapper
 - ▶ output: whatever
 - Both can do arbitrary computations on the input data as long as the basic structure is followed



MapReduce Framework



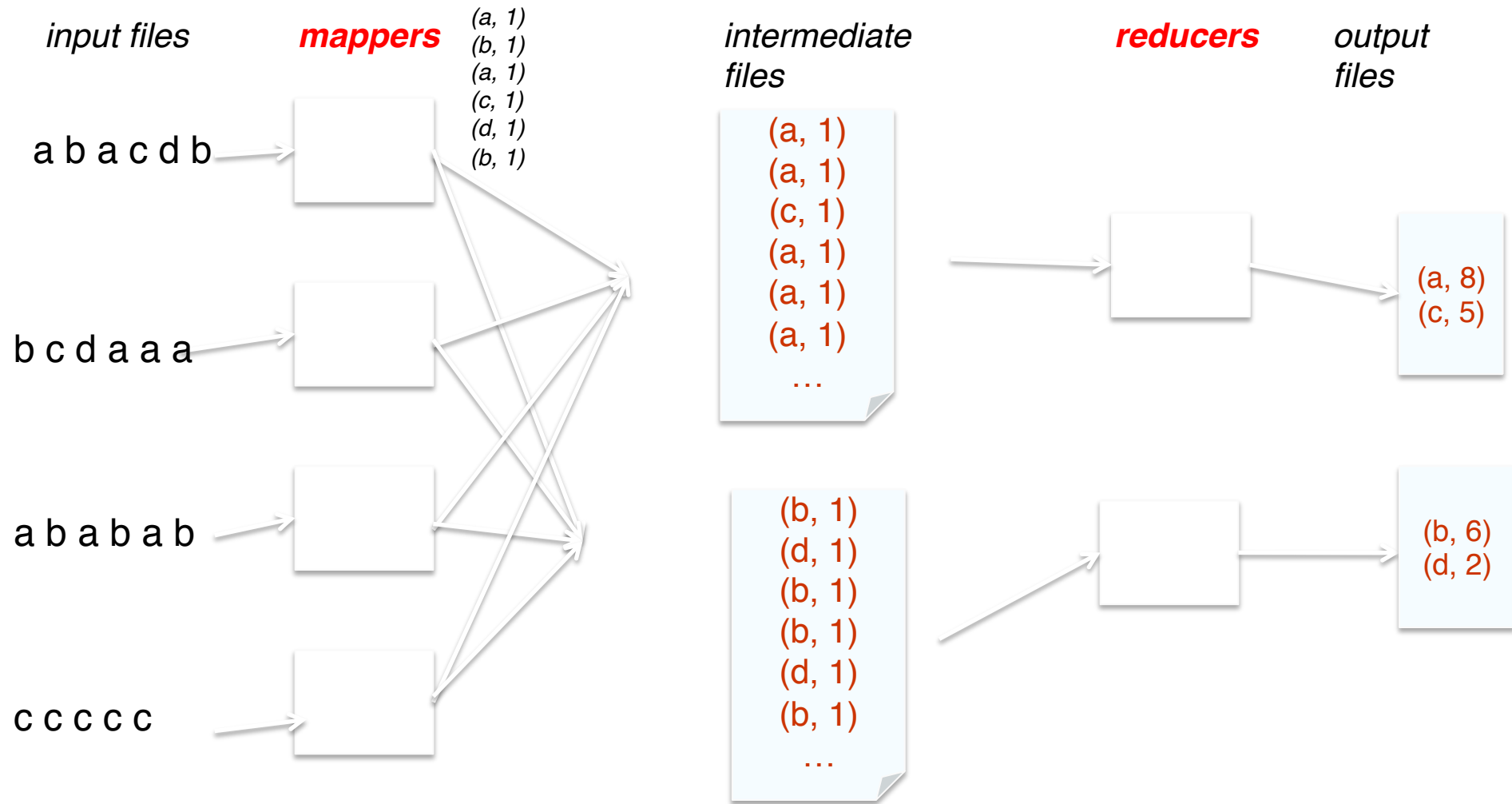


Word Count Example

```
map(String key, String value):  
    // key: document name  
    // value: document contents  
    for each word w in value:  
        EmitIntermediate(w, "1");  
  
reduce(String key, Iterator values):  
    // key: a word  
    // values: a list of counts  
    int result = 0;  
    for each v in values:  
        result += ParseInt(v);  
    Emit(AsString(result));
```

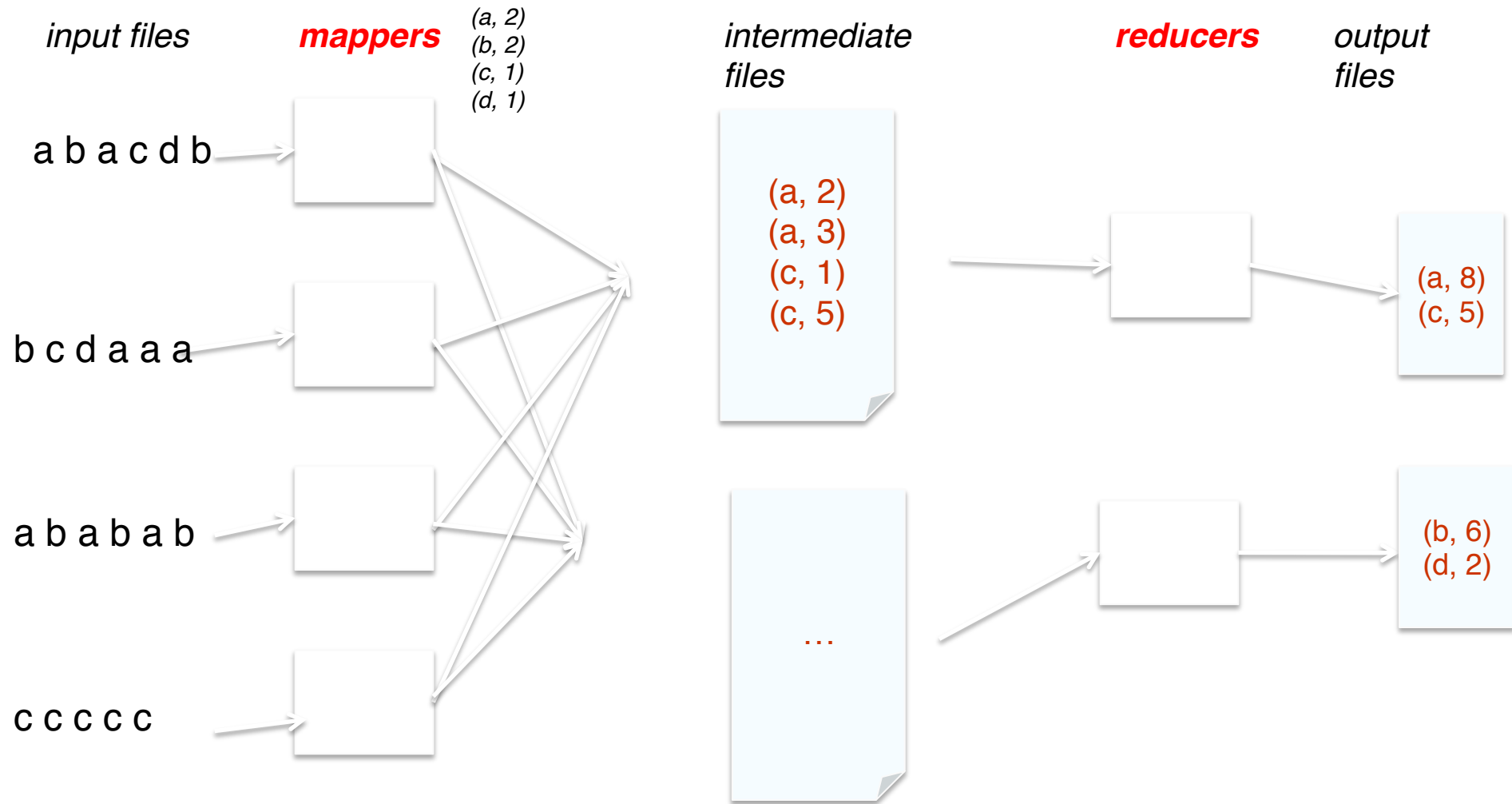


MapReduce Framework: Word Count





More Efficient Word Count



Called “mapper-side” combiner



MapReduce Framework

- Has been used within Google for:
 - Large-scale machine learning problems
 - Clustering problems for Google News etc..
 - Generating summary reports
 - Large-scale graph computations
- Also replaced the original tools for large-scale indexing
 - i.e., generating the inverted indexes etc.
 - runs as a sequence of 5 to 10 Mapreduce operations
- Hadoop:
 - Open-source implementation of Mapreduce
 - Supports many other technologies as well
 - Very widely used
 - Many startups focusing on providing Hadoop services, different points in the Hadoop/DB space etc...



Bigtable/Key-Value Stores

- MapReduce/Hadoop great for batch processing of data
 - Much ongoing work on efficiency, other programming frameworks (e.g., for graph analytics, scientific applications)
- There is another usecase
 - Very very large-scale web applications that need real-time access with few ms latencies
- Bigtable (open source implementation: HBase)
 - Think of it as a very large distributed hash table
 - Support “put” and “get” operations
 - ▶ With some additional support to deal with versions
- Much work on these systems
 - Issues with “consistency” and “performance” quite challenging



Key-Value Stores

- Some Interesting (somewhat old) numbers (<http://highscalability.com>)
 - Twitter: 177M tweets sent on 3/1/2011 (nothing special about the date), 572,000 accounts added on 3/12/2011
 - Dropbox: 1M files saved every 15 mins
 - Stackoverflow: 3M page views a day (Redis for caching)
 - Wordnik: 10 million API Requests a Day on MongoDB and Scala
 - Mollom: Killing Over 373 Million Spams at 100 Requests Per Second (Cassandra)
 - Facebook's New Real-time Messaging System: HBase to Store 135+ Billion Messages a Month
 - Reddit: 270 Million Page Views a Month in May 2010 (Memcache)
- How to support such scale?
 - Databases typically not fast enough
 - Facebook aims for 3-5ms response times



Issues

- Data Consistency, High Availability, and Low Latency hard to guarantee simultaneously
 - Impossible in some cases especially if networks can fail
- Distributed transactions
 - If a transaction spans multiple machines, what to do ?
 - Correct solution: Two-phase Commit
 - ▶ Multi-round protocol
 - ▶ Too high latencies
- Dealing with replication
 - Replication of data is a must
 - How to keep them updated?
 - ▶ Eager vs lazy replication
 - ▶ Significant impact on consistency and availability
- Many systems in this space sacrifice consistency



Systems

- Numerous systems designed in last 10 years that look very similar
 - Differences often subtle, and if not hard to pin down, hard to understand
 - Often the differences are about the implementations
- Often called key-value stores
 - The main provided functionality is that of a hashtable
- Some earlier solutions
 - Still very popular
 - ▶ Memcached + MySQL + Sharding
 - Sharding == partitioning
 - Store data in MySQL -- use Memcached to cache the data
 - ▶ Memcached not really a database, just a cache
 - ▶ All kinds of consistency issues
 - ▶ But... very very fast



Systems

■ MySQL + Memcached: End of an era? (High Scalability Blog)

- *“If you look at the early days of this blog, when web scalability was still in its heady bloom of youth, many of the articles had to do with leveraging MySQL and memcached. Exciting times. Shard MySQL to handle high write loads, cache objects in memcached to handle high read loads, and then write a lot of glue code to make it all work together. That was state of the art, that was how it was done. The architecture of many major sites still follow this pattern today, largely because with enough elbow grease, it works.”*

■ Digg moved to Cassandra in 2009; LinkedIn to Voldemort

■ Twitter moved to Cassandra recently

- “.. the rate of growth is accelerating.. a system in place based on shared mysql + memcache .. quickly becoming prohibitively costly (in terms of manpower) to operate.



Systems

- Tokyo, Redis
 - Very efficient key value stores
- BigTable (Google), HBase (Apache open source), Cassandra (original Facebook, open sourced), Voldemort (originally LinkedIn)...
 - At least in original iterations, focused on performance
 - Cassandra later developed more sophisticated {tunable} consistency (maybe others too)
- PNUTS (Yahoo!)
 - Focus on geographically distributed stuff
 - ▶ Easier to deal with some issues if we assume everything is a single data center
 - ▶ Support tunable consistency for reads: read-any, read-latest etc..
 - Form of master-slave replication
 - No real support for multi-record transactions



Systems

■ Megastore (Google)

- Built on top of BigTable -- powers Google App Engine
 - ▶ Full ACID using Paxos, replication, two-phase commit
- Supports notion of “entity groups”
 - ▶ e.g., all emails of a user is a single entity group
 - ▶ Transactions that span a single entity group are generally fine
 - ▶ Transactions that span multiple entity groups would use two-phase commit -- not preferred

■ MongoDB

- Perhaps the poster child of key-value NoSQL stores
- Very scalable
 - ▶ Document-oriented storage with JSON-style documents
 - ▶ JSON becoming more popular than XML as the interchange format
- Very loose consistency guarantees



In Summary...

■ Three key pieces of cloud computing

● Data centers

- ▶ Increasingly growing in numbers
- ▶ Many challenges in building them, maintaining them etc..

● Virtualization

● Programming frameworks

- ▶ Simplest (to explain): just use the virtual machines directly
 - But much harder to manage
- ▶ Using Hadoop or HBase (as appropriate) simplifies the programming quite a bit
 - But Hadoop is open source, and managing hadoop installations not much easier

■ Still many technical challenges to be solved



Cloud Computing: Outline

- Technologies behind cloud computing
 - Data centers
 - Virtualization
 - Programming Frameworks
- Some observations about the marketplace



Amazon Web Services

- Perhaps the best current solution to cloud computing
 - However alternatives become attractive depending on your needs
 - Current prices are very low and likely to remain that way

Small Instance – default*

1.7 GB memory
1 EC2 Compute Unit (1 virtual core with 1 EC2 Compute Unit)
160 GB instance storage
32-bit platform
I/O Performance: Moderate
API name: m1.small

Large Instance

7.5 GB memory
4 EC2 Compute Units (2 virtual cores with 2 EC2 Compute Units each)
850 GB instance storage
64-bit platform
I/O Performance: High
API name: m1.large

Extra Large Instance

15 GB memory
8 EC2 Compute Units (4 virtual cores with 2 EC2 Compute Units each)
1,690 GB instance storage
64-bit platform
I/O Performance: High
API name: m1.xlarge

Region:	US East (Virginia)	
	Linux/UNIX Usage	Windows Usage
Standard On-Demand Instances		
Small (Default)	\$0.085 per hour	\$0.12 per hour
Large	\$0.34 per hour	\$0.48 per hour
Extra Large	\$0.68 per hour	\$0.96 per hour
Micro On-Demand Instances		
Micro	\$0.02 per hour	\$0.03 per hour
Hi-Memory On-Demand Instances		
Extra Large	\$0.50 per hour	\$0.62 per hour
Double Extra Large	\$1.00 per hour	\$1.24 per hour
Quadruple Extra Large	\$2.00 per hour	\$2.48 per hour
Hi-CPU On-Demand Instances		
Medium	\$0.17 per hour	\$0.29 per hour
Extra Large	\$0.68 per hour	\$1.16 per hour
Cluster Compute Instances		
Quadruple Extra Large	\$1.60 per hour	N/A*
Cluster GPU Instances		
Quadruple Extra Large	\$2.10 per hour	N/A*
* Windows® is not currently available for Cluster Compute or Cluster GPU Instances		



Google App Engine

- A very nice solution to build your websites on top of Google infrastructure
 - e.g., <http://cidrassgn.appspot.com/>
 - No virtual machines or any other way to access the computing, just through a web app

- Recently (two weeks ago) increased their pricing quite a bit
 - A lot of developers are very unhappy
 - [Google Groups Thread](#)
 - Also serves as a very nice reference to competing services, differences between them etc...
 - Also, bunch of discussion on how people spent optimizing their apps for the “wrong” metrics (i.e., Google is starting to charge for things they weren’t)

Topics

- Object Oriented, Object Relational
- Client-server, Parallel, Distributed Systems
- OLAP/Data Warehouses
- Information Retrieval
- Cloud Computing
 - ★ Data centers, Map-reduce, NoSQL Systems

Motivation

- Relational model:
 - ★ Clean and simple
 - ★ Great for much enterprise data
 - ★ But lot of applications where not *sufficiently rich*
 - Multimedia, CAD, for storing set data etc
- Object-oriented models in programming languages
 - ★ Complicated, but very useful
 - Smalltalk, C++, now Java
 - ★ Allow
 - Complex data types
 - Inheritance
 - Encapsulation
- People wanted to manage objects in databases.

History

- In the 1980's and 90's, DB researchers recognized benefits of objects.
- Two research thrusts:
 - ★ OODBMS: extend C++ with transactionally persistent objects
 - Niche Market
 - CAD etc
 - ★ ORDBMS: extend Relational DBs with object features
 - Much more common
 - Efficiency + Extensibility
 - SQL:99 support
- Postgres – First ORDBMS
 - ★ Berkeley research project
 - ★ Became Illustra, became Informix, bought by IBM



Object-Relational Data Models

- Extend the relational data model by including object orientation and constructs to deal with added data types.
- Allow attributes of tuples to have complex types, including non-atomic values such as nested relations.
- Preserve relational foundations, in particular the declarative access to data, while extending modeling power.
- Upward compatibility with existing relational languages.



Structured Types and Inheritance in SQL

- **Structured types** (a.k.a. **user-defined types**) can be declared and used in SQL

create type *Name* **as**

 (*firstname* **varchar**(20),
 lastname **varchar**(20))
 final

create type *Address* **as**

 (*street* **varchar**(20),
 city **varchar**(20),
 zipcode **varchar**(20))
 not final

- Note: **final** and **not final** indicate whether subtypes can be created
- Structured types can be used to create tables with composite attributes
 - create table** *person* (
 name *Name*,
 address *Address*,
 dateOfBirth **date**)
- Dot notation used to reference components: *name.firstname*



Structured Types (cont.)

■ User-defined row types

```
create type PersonType as (  
    name Name,  
    address Address,  
    dateOfBirth date)  
not final
```

■ Can then create a table whose rows are a user-defined type

```
create table customer of CustomerType
```

■ Alternative using **unnamed row types**.

```
create table person_r(  
    name    row(firstname varchar(20),  
                lastname varchar(20)),  
    address row(street    varchar(20),  
                city       varchar(20),  
                zipcode   varchar(20)),  
    dateOfBirth date)
```



Methods

- Can add a method declaration with a structured type.

method *ageOnDate* (*onDate* **date**)

returns **interval year**

- Method body is given separately.

create instance method *ageOnDate* (*onDate* **date**)

returns **interval year**

for *CustomerType*

begin

return *onDate* - **self.dateOfBirth**;

end

- We can now find the age of each customer:

select *name.lastname*, *ageOnDate* (**current_date**)

from *customer*



Type Inheritance

- Suppose that we have the following type definition for people:

```
create type Person  
  (name varchar(20),  
   address varchar(20))
```

- Using inheritance to define the student and teacher types

```
create type Student  
under Person  
  (degree varchar(20),  
   department varchar(20))
```



```
create type Teacher  
under Person  
  (salary integer,  
   department varchar(20))
```

- Subtypes can redefine methods by using **overriding method** in place of **method** in the method declaration



Array and Multiset Types in SQL

- Example of array and multiset declaration:

create type *Publisher* **as**

 (*name* **varchar**(20),
 branch **varchar**(20));

create type *Book* **as**

 (*title* **varchar**(20),
 author_array **varchar**(20) **array** [10],
 pub_date **date**,
 publisher *Publisher*,
 keyword-set **varchar**(20) **multiset**);

create table *books* **of** *Book*;



Creation of Collection Values

- Array construction
array ['Silberschatz', `Korth`, `Sudarshan']
- Multisets
multiset ['computer', 'database', 'SQL']
- To create a tuple of the type defined by the books relation:
('Compilers', **array**[`Smith`, `Jones`],
new Publisher (`McGraw-Hill`, `New York`),
multiset [`parsing`, `analysis'])
- To insert the preceding tuple into the relation books
insert into *books*
values
('Compilers', **array**[`Smith`, `Jones`],
new Publisher (`McGraw-Hill`, `New York`),
multiset [`parsing`, `analysis']);



Querying Collection-Valued Attributes

- To find all books that have the word “database” as a keyword,
select *title*
from *books*
where ‘database’ **in** (**unnest**(*keyword-set*))
- We can access individual elements of an array by using indices
 - E.g.: If we know that a particular book has three authors, we could write:
select *author_array*[1], *author_array*[2], *author_array*[3]
from *books*
where *title* = ‘Database System Concepts’
- To get a relation containing pairs of the form “title, author_name” for each book and each author of the book
select *B.title*, *A.author*
from *books as B*, **unnest** (*B.author_array*) **as** *A (author)*
- To retain ordering information we add a **with ordinality** clause
select *B.title*, *A.author*, *A.position*
from *books as B*, **unnest** (*B.author_array*) **with ordinality as**
A (author, position)



Path Expressions

- Find the names and addresses of the heads of all departments:
select *head* → *name*, *head* → *address*
from *departments*
- An expression such as “*head* → *name*” is called a **path expression**
- Path expressions help avoid explicit joins
 - If department head were not a reference, a join of *departments* with *people* would be required to get at the address
 - Makes expressing the query much easier for the user

An Alternative: OODBMS

- Persistent OO programming
 - ★ Imagine declaring a Java object to be “persistent”
 - ★ Everything reachable from that object will also be persistent
 - ★ You then write plain old Java code, and all changes to the persistent objects are stored in a database
 - ★ When you run the program again, those persistent objects have the same values they used to have!
- Solves the “impedance mismatch” between programming languages and query languages
 - ★ E.g. converting between Java and SQL types, handling rowsets, etc.
 - ★ But this programming style doesn't support declarative queries
 - For this reason (??), OODBMSs haven't proven popular
- OQL: A declarative language for OODBMSs
 - ★ Was only implemented by one vendor in France (Altair)

OODBMS

- Currently a Niche Market
 - ★ Engineering, spatial databases, physics etc...
- Main issues:
 - ★ Navigational access
 - Programs specify go to this object, follow this pointer
 - ★ Not declarative
- Though advantageous when you know exactly what you want, not a good idea in general
 - ★ Kinda similar argument as *network databases vs relational databases*



Comparison of O-O and O-R Databases

■ Relational systems

- simple data types, powerful query languages, high protection.

■ Persistent-programming-language-based OODBs

- complex data types, integration with programming language, high performance.

■ Object-relational systems

- complex data types, powerful query languages, high protection.

■ Object-relational mapping systems

- complex data types integrated with programming language, but built as a layer on top of a relational database system

■ Note: Many real systems blur these boundaries

- E.g. persistent programming language built as a wrapper on a relational database offers first two benefits, but may have poor performance.

Summary, cont.

■ ORDBMS offers many new features

- ★ but not clear how to use them!
- ★ schema design techniques not well understood
 - No good logical design theory for non-1st-normal-form!
- ★ query processing techniques still in research phase
 - a moving target for OR DBA's!

■ OODBMS

- ★ Has its advantages
- ★ Niche market
- ★ Lot of similarities to XML as well...

Topics

- Object Oriented, Object Relational
- Client-server, Parallel, Distributed Systems
- OLAP/Data Warehouses
- Information Retrieval
- Cloud Computing
 - ★ Data centers, Map-reduce, NoSQL Systems

OLAP

■ On-line Analytical Processing

■ Why ?

★ Exploratory analysis

- Interactive
- Different queries than typical SPJ SQL queries

★ Data CUBE

- A summary structure used for this purpose
 - E.g. *give me total sales by zipcode; now show me total sales by customer employment category*
- Much much faster than using SQL queries against the raw data
 - The tables are *huge*

■ Applications:

- ★ Sales reporting, Marketing, Forecasting etc etc

Data Warehouses

- A repository of integrated information for querying and analysis purposes
- A (usually) stand-alone system that integrates data from everywhere
 - ★ Read-only, typically not kept up-to-date with the *real* data
 - ★ Geared toward business analytics, data mining etc...
 - ★ HUGE market today
- Heavily optimized
 - ★ Specialized query processing and indexing techniques are used
 - ★ High emphasis on pre-computed data structures like summary tables, **data cubes**
- Analysis cycle:
 - ★ Extract data from databases with queries, visualize/analyze with desktop tools
 - ★ E.g., Tableau

Data Warehouses

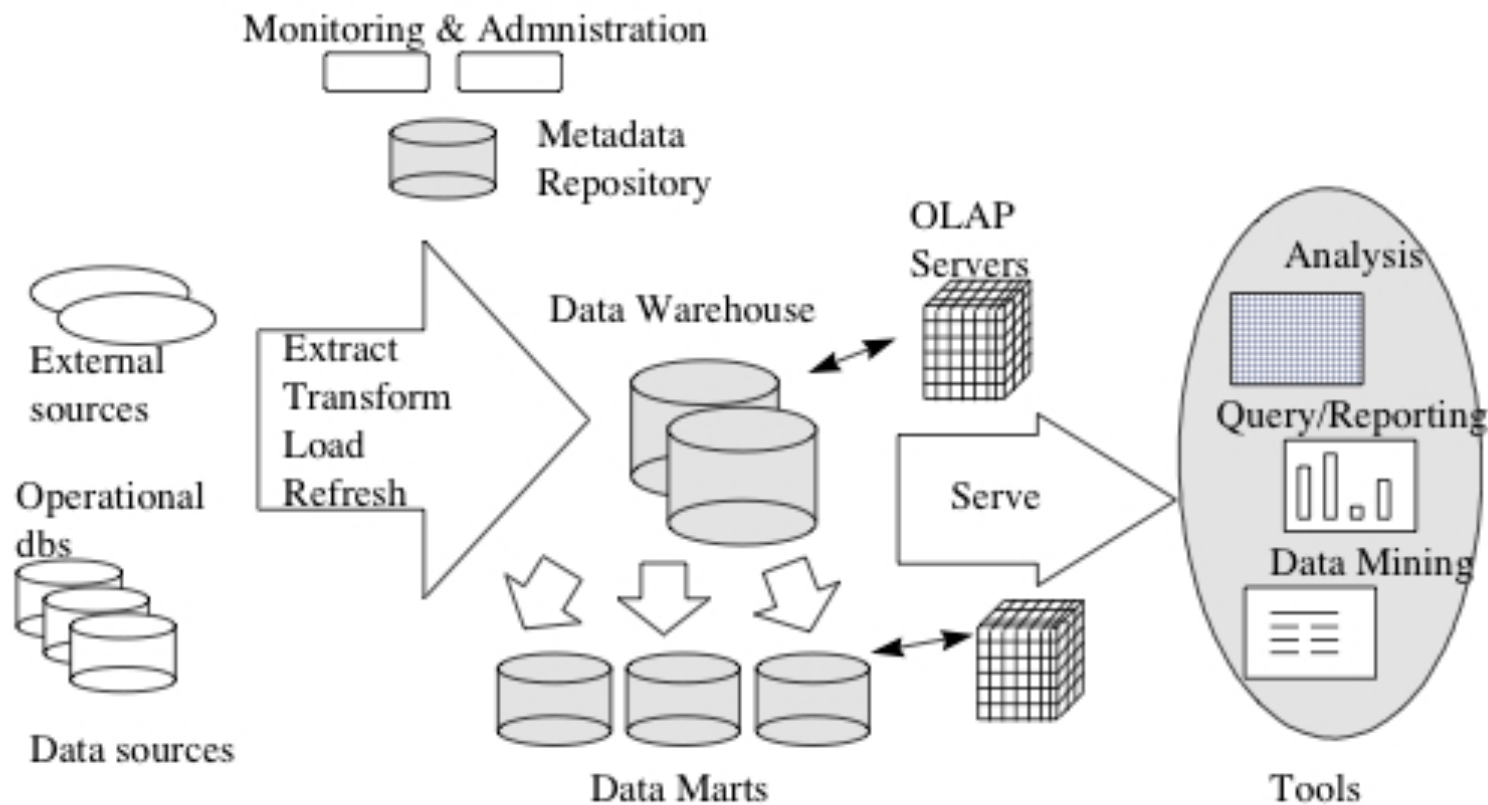


Figure 1. Data Warehousing Architecture

Data Warehouses

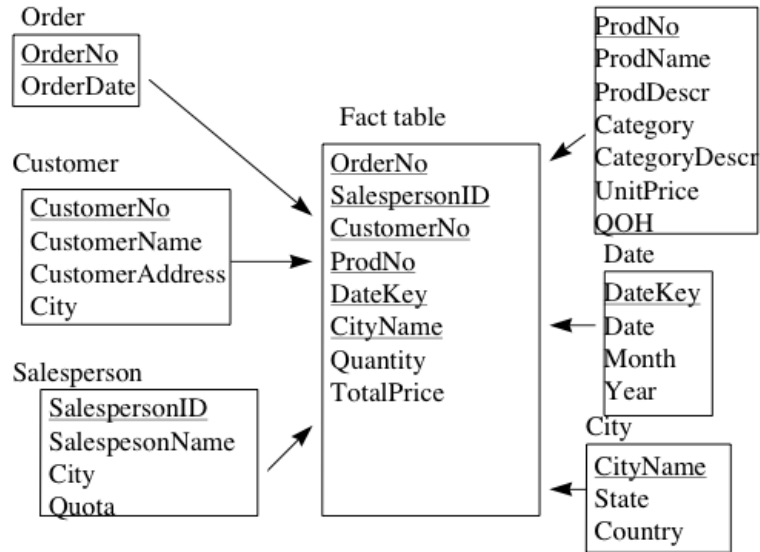


Figure 3. A Star Schema.

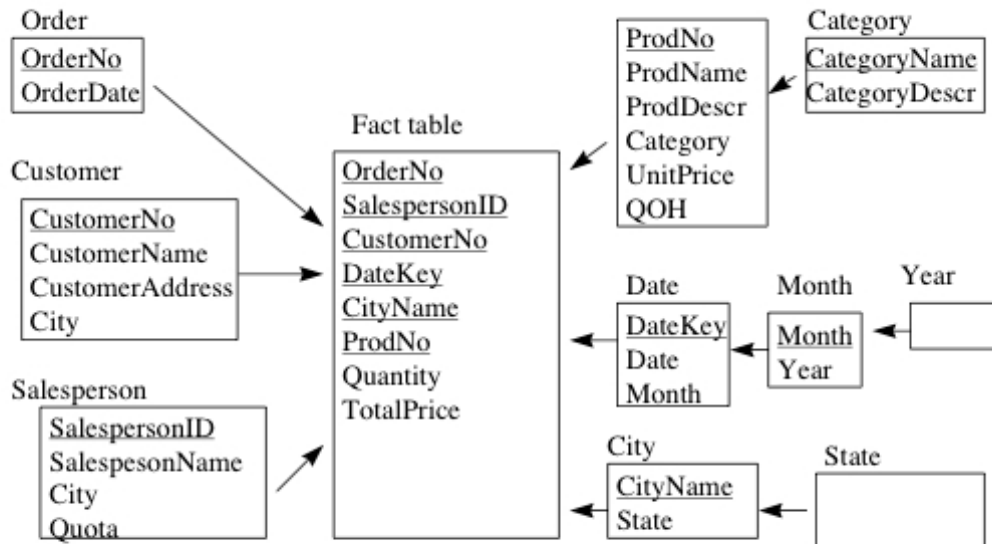


Figure 4. A Snowflake Schema.

Query processing algorithms heavily optimized for these types of schemas

Many queries of the type:

Selections on dimension tables

(e.g., state = 'MD')

Join fact table with dimension tables

Aggregate on a "measure" attribute
(e.g., Quantity, TotalPrice)

For example:

```
select c_city, o_year, SUM(quantity)
from Fact, Customer, Product
where p_category = 'Tablet';
```


Need Generalized SQL Groupbys

■ drill-down and roll-up

Table 3: Sales Roll Up by Model by Year by Color

Model	Year	Color	Sales by Model by Year by Color	Sales by Model by Year	Sales by Model
Chevy	1994	black	50		
		white	40		
				90	
	1995	black	85		
		white	115		
				200	
					290

Not relational
(null values in the keys)

Table 4: Sales Summary

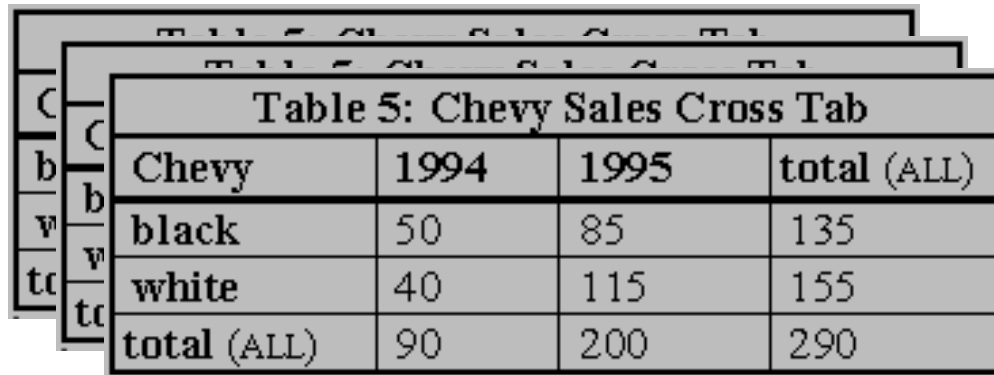
Model	Year	Color	Units
Chevy	1994	black	50
Chevy	1994	white	40
Chevy	1994	ALL	90
Chevy	1995	black	85
Chevy	1995	white	115
Chevy	1995	ALL	200
Chevy	ALL	ALL	290

```

SELECT Model, ALL, ALL, SUM(Sales)
  FROM Sales
 WHERE Model = 'Chevy'
 GROUP BY Model
UNION
SELECT Model, Year, ALL, SUM(Sales)
  FROM Sales
 WHERE Model = 'Chevy'
 GROUP BY Model, Year
UNION
SELECT Model, Year, Color, SUM(Sales)
  FROM Sales
 WHERE Model = 'Chevy'
 GROUP BY Model, Year, Color;
    
```

More problems with Groubys

- roll-up is asymmetric (e.g. does not aggregate by year or by color alone)
- cross-tabulation (spreadsheets)



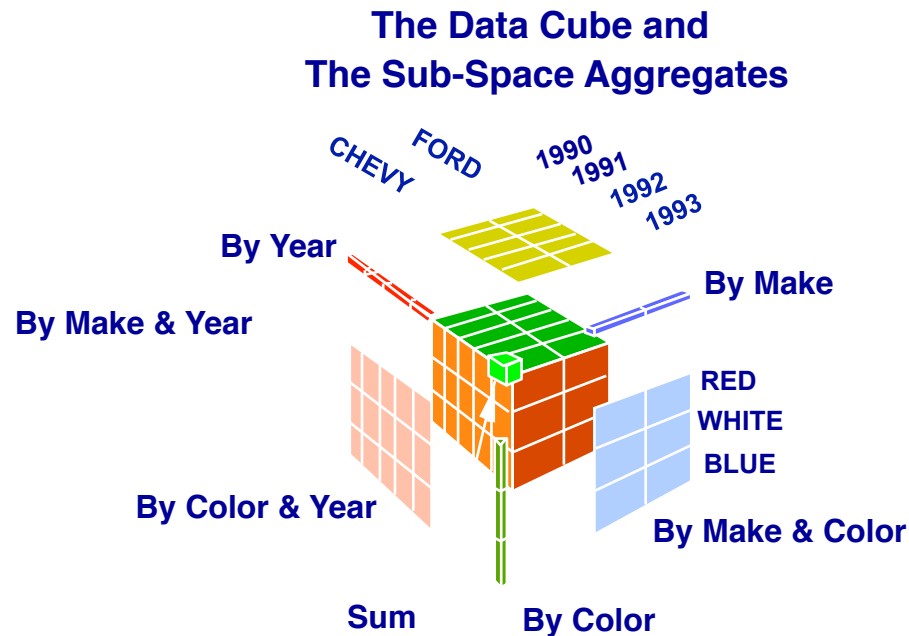
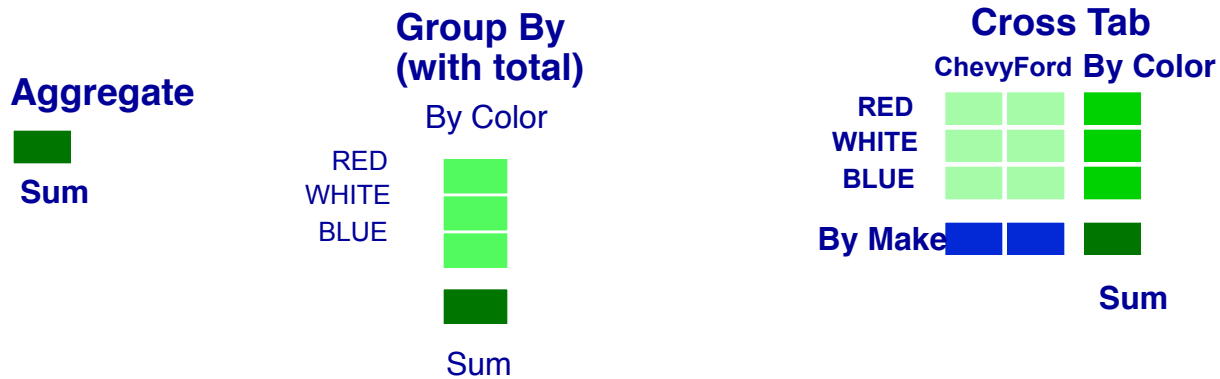
The image shows three overlapping spreadsheets. The top spreadsheet is titled 'Table 5: Chevy Sales Cross Tab'. It contains a table with the following data:

Chevy	1994	1995	total (ALL)
black	50	85	135
white	40	115	155
total (ALL)	90	200	290

- even if SQL syntax can be devised, a 6D cross-tab requires 64 groupby queries to generate it and 64 scans and sorts of the data
- ◆ most of these are not relational expressions but are in many report writers

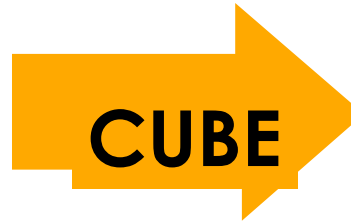
CUBE:

A Relational Aggregate Operator Generalizing Group By



An Example

SALES			
Model	Year	Color	Sales
Chevy	1990	red	5
Chevy	1990	white	87
Chevy	1990	blue	62
Chevy	1991	red	54
Chevy	1991	white	95
Chevy	1991	blue	49
Chevy	1992	red	31
Chevy	1992	white	54
Chevy	1992	blue	71
Ford	1990	red	64
Ford	1990	white	62
Ford	1990	blue	63
Ford	1991	red	52
Ford	1991	white	9
Ford	1991	blue	55
Ford	1992	red	27
Ford	1992	white	62
Ford	1992	blue	39



DATA CUBE			
Model	Year	Color	Sales
ALL	ALL	ALL	942
chevy	ALL	ALL	510
ford	ALL	ALL	432
ALL	1990	ALL	343
ALL	1991	ALL	314
ALL	1992	ALL	285
ALL	ALL	red	165
ALL	ALL	white	273
ALL	ALL	blue	339
chevy	1990	ALL	154
chevy	1991	ALL	199
chevy	1992	ALL	157
ford	1990	ALL	189
ford	1991	ALL	116
ford	1992	ALL	128
chevy	ALL	red	91
chevy	ALL	white	236
chevy	ALL	blue	183
ford	ALL	red	144
ford	ALL	white	133
ford	ALL	blue	156
ALL	1990	red	69
ALL	1990	white	149
ALL	1990	blue	125
ALL	1991	red	107
ALL	1991	white	104
ALL	1991	blue	104
ALL	1992	red	59
ALL	1992	white	116
ALL	1992	blue	110

Data Mining

■ Searching for patterns in data

- ★ Typically done in data warehouses

■ Association Rules:

- ★ When a customer buys X, she also typically buys Y

- ★ Use ?

- Move X and Y together in supermarkets

- ★ A customer buys a lot of shirts

- Send him a catalogue of shirts

- ★ Patterns are not always obvious

- Classic example: It was observed that men tend to buy *beer* and *diapers* together (may be an urban legend)

■ Other types of mining

- ★ Classification

- ★ Decision Trees

Data Warehouses

- Data analytics a major industry right now, and likely to grow in near future
 - ★ BIG Data !!
 - ★ Extracting (actionable) knowledge from data really critical
 - Especially in real-time
- Some key technologies:
 - ★ Parallelism – pretty much required
 - ★ Column-oriented design
 - Lay out the data column-by-column, rather than row-by-row
 - ★ Heavy pre-computation (like Cubes)
 - ★ New types of indexes
 - Focusing on bitmap representations
 - ★ Heavy compression
 - ★ Map-reduce??

Topics

- Object Oriented, Object Relational
- Client-server, Parallel, Distributed Systems
- OLAP/Data Warehouses
- Information Retrieval
- Cloud Computing
 - ★ Data centers, Map-reduce, NoSQL Systems

Information Retrieval

- Relational DB == Structured data
- Information Retrieval == Unstructured data
- Evolved independently of each other
 - ★ Still very little interaction between the two
- Goal: Searching within documents
 - ★ Queries are different; typically a list of words, not SQL
- E.g. Web searching
 - ★ If you just look for documents containing the words, millions of them
 - Mostly useless
- Ranking:
 - ★ This is the key in IR
 - ★ Many different ways to do it
 - E.g. something that takes into account *term frequencies*
 - ★ Pagerank (from Google) seems to work best for Web.



Relevance Ranking Using Terms

■ **TF-IDF** (Term frequency/Inverse Document frequency) ranking:

- Let $n(d)$ = number of terms in the document d
- $n(d, t)$ = number of occurrences of term t in the document d .
- Relevance of a document d to a term t

$$TF(d, t) = \log \left(1 + \frac{n(d, t)}{n(d)} \right)$$

- ▶ The log factor is to avoid excessive weight to frequent terms
- Relevance of document to query Q

$$r(d, Q) = \sum_{t \in Q} \frac{TF(d, t)}{n(t)}$$



PageRank

- The probability that a random surfer (who follows links randomly) will end up at a particular page
 - **Intuitively:** Higher the probability, the more important the page
- Surfer model:
 - Choose a random page to visit with probability “alpha”
 - If the number of outgoing edges = n , then visit one of those pages with probability $(1 - \alpha)/n$

