Distributed Databases and NOSQL

Introduction to Databases

CompSci 316 Spring 2017

DUKE
COMPUTER SCIENCE

Announcements (Mon., Apr. 24)

- Homework #4 due today (Monday, April 24, 11:55 pm)
- Project
  - final report draft due today -- Monday, April 24, 11:59 pm
  - code due on Wednesday -- April 26, 11:59 pm
  - See all announcements about project report and demo on piazza
- Please bring a computer to class on Wednesday
  - We will take a 5-10 mins break for filling out course evaluations as advised by the Office of Assessments
- Google Cloud code
  - Please redeem your code asap, by May 11
- Final Exam
  - May 2 (next Tuesday), 2-5 pm, in class
  - Everything covered in the class (up to last lecture) is included
  - If you need special accommodation, please email me

Announcements (Mon., Apr. 24)

- Final Project Report
  - should be "comprehensive"
  - if you had more material in MS1 or MS2, please include them
    - not the "work in progress" part

- The "overview of your application" part should include
  - description of the user interface of your system – screenshots, features, how the user will be able to interact with the system
  - Sample execution (input and output)
  - Your approach (algorithm, indexes, storage, optimization, normalization)
  - Any interesting observation

Where are we now?

We learnt

- ✓ Relational Model, Query Languages, and Database Design
  - ✓ SQL
  - ✓ RA
  - ✓ E/R diagram
  - ✓ Normalization
- ✓ DBMS Internals
  - ✓ Storage
  - ✓ Indexing
  - ✓ Query Evaluation
  - ✓ External sort
  - ✓ Join Algorithms
  - ✓ Query Optimization
- ✓ Transactions
  - ✓ Basic concepts
  - ✓ Concurrency control
  - ✓ Recovery
- ✓ XML
  - ✓ DTD and XML schema
  - ✓ XPath and XQuery
  - ✓ Relational Mapping
- ✓ Advanced/Research topics overview
  - ✓ Parallel DBMS
  - ✓ Map Reduce
  - ✓ Data Mining
  - ✓ Data Warehousing

- Today
  - Distributed DBMS
  - NOSQL
- Next lecture
  - Overview of other areas database researchers work on
  - Practice problems for the final

Parallel vs. Distributed DBMS

Parallel DBMS

- Parallelization of various operations
  - e.g. loading data, building indexes, evaluating queries
- Data may or may not be distributed initially
- Distribution is governed by performance consideration

Distributed DBMS

- Data is physically stored across different sites
  - Each site is typically managed by an independent DBMS
- Location of data and autonomy of sites have an impact on Query opt., Conc. Control and recovery
- Also governed by other factors:
  - increased availability for system crash
  - local ownership and access

Distributed Databases

Architecture
Data Storage
Query Execution
Transactions

Two desired properties and recent trends

- Data is stored at several sites, each managed by a DBMS that can run independently

1. Distributed Data Independence
   - Users should not have to know where data is located
2. Distributed Transaction Atomicity
   - Users should be able to write transactions accessing multiple sites just like local transactions

- These two properties are in general desirable, but not always efficiently achievable
  - e.g. when sites are connected by a slow long-distance network
- Even sometimes not desirable for globally distributed sites
  - too much administrative overhead of making location of data transparent (not visible to the user)
- Therefore not always supported
  - Users have to be aware of where data is located

Distributed DBMS Architecture

- Three alternative approaches

1. Client-Server
   - One or more client (e.g. personal computer) and one or more server processes (e.g. a mainframe)
   - Clients are responsible for user interfaces, Server manages data and executes queries
2. Collaborating Server
   - Queries can be submitted and can span multiple sites
   - No distinction between clients and servers
3. Middleware
   - need just one db server (called middleware) capable of managing queries and transactions spanning multiple servers
   - the remaining servers can handle only the local queries and transactions
   - can integrate legacy systems with limited flexibility and power

Storing Data in a Distributed DBMS

- Relations are stored across several sites
- Accessing data at a remote site incurs message-passing costs
- To reduce this overhead, a single relation may be partitioned or fragmented across several sites
  - typically at sites where they are most often accessed
- The data can be replicated as well
  - when the relation is in high demand

Fragmentation

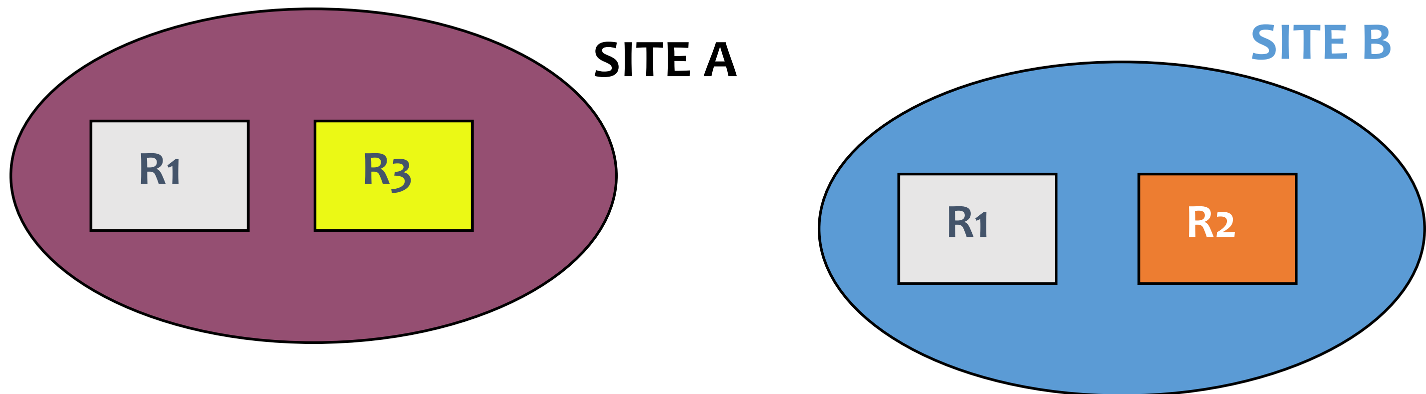
- Break a relation into smaller relations or fragments
  - store them in different sites as needed

TID					
t1					
t2					
t3					
t4					

- Horizontal:
  - Usually disjoint
  - Can often be identified by a selection query (employees in a city – locality of reference)
  - To retrieve the full relation, need a union
- Vertical:
  - Identified by projection queries
  - Typically unique TIDs added to each tuple
  - TIDs replicated in each fragments
  - Ensures that we have a Lossless Join

Replication

- When we store several copies of a relation or relation fragments
  - can be replicated at one or more sites
  - e.g. R is fragmented into R1, R2, R3; one copy of R2, R3; but two copies at R1 at two sites
- Advantages
  - Gives increased availability – e.g. when a site or communication link goes down
  - Faster query evaluation – e.g. using a local copy
- Synchronous and Asynchronous (later)
  - Vary in how current different copies are when a relation is modified



Distributed Query Processing: Non-Join Distributed Queries

```
SELECT AVG(S.age)
FROM Sailors S
WHERE S.rating > 3
  AND S.rating < 7
```

tid	sid	sname	rating	age
T1			4	
T2			5	
T3			9	

T1: stored at Shanghai

T2, T3: stored at Tokyo

- Horizontally Fragmented: Tuples with rating < 5 at Shanghai, >= 5 at Tokyo.
  - Must compute SUM(age), COUNT(age) at both sites.
  - If WHERE contained just S.rating > 6, just one site
- Vertically Fragmented: sid and rating at Shanghai, sname and age at Tokyo, tid at both.
  - Must reconstruct relation by join on tid, then evaluate the query
  - if no tid, decomposition would be lossy
- Replicated: Sailors copies at both sites.
  - Choice of site based on local costs (e.g. index), shipping costs

Joins in a Distributed DBMS

- Can be very expensive if relations are stored at different sites
- Semi-join (we will cover only this)
- Other join methods:
  - Fetch as needed
  - Ship to one site
  - Bloom join (similar approach to semi-join but uses hashing)

LONDON

500 pages

PARIS

1000 pages

Semijoin -1/2

- Suppose want to ship R to London and then do join with S at London. Instead,

1. At London, project S onto join columns and ship this to Paris
   - Here foreign keys, but could be arbitrary join
2. At Paris, join S-projection with R
   - Result is called reduction of Reserves w.r.t. Sailors (only these tuples are needed)
3. Ship reduction of R to back to London
4. At London, join S with reduction of R

LONDON

500 pages

PARIS

1000 pages

Semijoin – 2/2

- Tradeoff the cost of computing and shipping projection for cost of shipping full R relation
- Especially useful if there is a selection on Sailors, and answer desired at London

LONDON

500 pages

PARIS

1000 pages

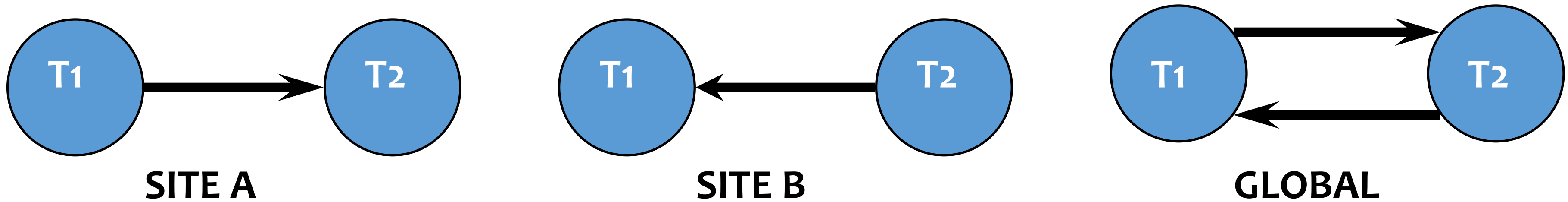
Updating Distributed Data

- Synchronous Replication:
  - All copies of a modified relation (or fragment) must be updated before the modifying transaction commits
  - Data distribution is made totally "transparent" (not visible!) to users
  - Before an update transaction can commit, it must obtain locks on all modified copies – Slow!
  - By Majority voting or "read any write all"
- Asynchronous Replication:
  - Copies of a modified relation are only periodically updated; different copies may get out of sync in the meantime
  - Users must be aware of data distribution
  - More efficient – many current products follow this approach
  - Update "master" copy and propagate (primary site), or update "any" copy and propagate (peer to peer)

Distributed Transactions

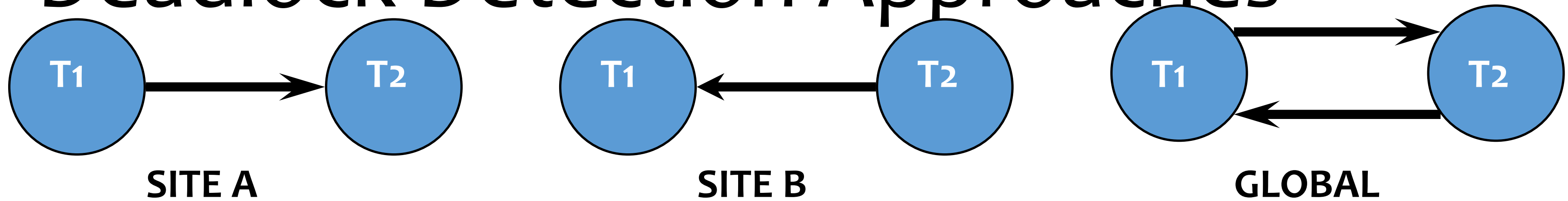
- Distributed CC
  - How can locks for objects stored across several sites be managed?
  - How can deadlocks be detected in a distributed database?
- Distributed Recovery
  - When a transaction commits, all its actions, across all the sites at which is executes must persist
  - When a transaction aborts, none of its actions must be allowed to persist

Distributed Deadlock Detection



- Locking can happen:
  - Centrally – one site manages all locks
  - Primary site – primary copy is locked
  - Distributed – by different sites storing a copy
- Each site maintains a local waits-for graph
- A global deadlock might exist even if the local graphs contain no cycles
- Further, phantom deadlocks may be created while communicating
  - due to delay in propagating local information
  - might lead to unnecessary aborts

Three Distributed Deadlock Detection Approaches



1. Centralized
   - send all local graphs to one site periodically
   - A global waits-for graph is generated
2. Hierarchical
   - organize sites into a hierarchy and send local graphs to parent in the hierarchy
   - e.g. sites (every 10 sec)-> sites in a state (every min)-> sites in a country (every 10 min) -> global waits for graph
   - intuition: more deadlocks are likely across closely related sites
3. Timeout
   - abort transaction if it waits too long (low overhead)

Distributed Recovery

- Two new issues:
  - New kinds of failure, e.g., links and remote sites
  - If “sub-transactions” of a transaction execute at different sites, all or none must commit
  - Need a commit protocol to achieve this
  - Most widely used: Two Phase Commit (2PC)

- A log is maintained at each site
  - as in a centralized DBMS
  - commit protocol actions are additionally logged

Two-Phase Commit (2PC)

- Site at which transaction originates is coordinator
- Other sites at which it executes are subordinates
    - w.r.t. coordinarion of this transaction
- Two rounds of communication (overview only)
    - Phase 1:
        - Prepare from Coordinator to all subs asking if ready to commit
        - Yes/No – from subs to coordinator
        - To commit, all subs must say yes
    - Phase 2:
        - If yes from all subs, coordinator sends out Commit
        - Subs do so, and send back ack
    - Before each message sent, the log is updated with the decision
    - If time out for the next message, ping other machines/restart/abort according to the state of the log

NoSQL

Leverage the NoSQL boom

- Optional reading:
  - Cattell's paper (2010-11)
  - Warning! some info will be outdated
  - see webpage http://cattell.net/datastores/ for updates and more pointers

So far -- RDBMS

- Relational Data Model
- Relational Database Systems (RDBMS)
- RDBMSs have
  - a complete pre-defined fixed schema
  - a SQL interface
  - and ACID transactions

NOSQL

- Many of the new systems are referred to as “NoSQL” data stores
  - MongoDB, CouchDB, VoltDB, Dynamo, Membase, ….

- NoSQL stands for “Not Only SQL” or “Not Relational”
  - not entirely agreed upon

- NoSQL = “new” database systems
  - not typically RDBMS
  - relax on some requirements, gain efficiency and scalability

- New systems choose to use/not use several concepts we learnt so far
  - e.g. System X does not use locks but use multi-version CC (MVCC) or,
  - System Y uses asynchronous replication

Applications of New Systems

- Designed to scale simple "OLTP"-style application loads
  - to do updates as well as reads
  - in contrast to traditional DBMSs and data warehouses
  - to provide good horizontal scalability for simple read/write database operations distributed over many servers
- Originally motivated by Web 2.0 applications
  - these systems are designed to scale to thousands or millions of users

NoSQL: Six Key Features

1. the ability to horizontally scale "simple operations" throughput over many servers
2. the ability to replicate and to distribute (partition) data over many servers
3. a simple call level interface or protocol (in contrast to SQL binding)
4. a weaker concurrency model than the ACID transactions of most relational (SQL) database systems
5. efficient use of distributed indexes and RAM for data storage
6. the ability to dynamically add new attributes to data records

BASE (not ACID ☺)

- Recall ACID for RDBMS desired properties of transactions:
  - Atomicity, Consistency, Isolation, and Durability

- NOSQL systems typically do not provide ACID

- Basically Available
- Soft state
- Eventually consistent

ACID vs. BASE

- The idea is that by giving up ACID constraints, one can achieve much higher performance and scalability

- The systems differ in how much they give up
  - e.g. most of the systems call themselves "eventually consistent", meaning that updates are eventually propagated to all nodes
  - but many of them provide mechanisms for some degree of consistency, such as multi-version concurrency control (MVCC)

“CAP” Theorem

- Often Eric Brewer’s CAP theorem cited for NoSQL
- A system can have only two out of three of the following properties:
  - Consistency,
  - Availability
  - Partition-tolerance
- The NoSQL systems generally give up consistency
  - However, the trade-offs are complex

What is different in NOSQL systems

- When you study a new NOSQL system, notice how it differs from RDBMS in terms of

1. Concurrency Control
2. Data Storage Medium
3. Replication
4. Transactions

Choices in NOSQL systems:
1. Concurrency Control

a) Locks
  - some systems provide one-user-at-a-time read or update locks
  - MongoDB provides locking at a field level

b) MVCC

c) None
  - do not provide atomicity
  - multiple users can edit in parallel
  - no guarantee which version you will read

d) ACID
  - pre-analyze transactions to avoid conflicts
  - no deadlocks and no waits on locks

Choices in NOSQL systems: 2. Data Storage Medium

a) Storage in RAM
   - snapshots or replication to disk
   - poor performance when overflows RAM

b) Disk storage
   - caching in RAM

Choices in NOSQL systems:
3. Replication

- whether mirror copies are always in sync

a) Synchronous

b) Asynchronous
   - faster, but updates may be lost in a crash

c) Both
   - local copies synchronously, remote copies asynchronously

Choices in NOSQL systems: 4. Transaction Mechanisms

a) support
b) do not support
c) in between
   - support local transactions only within a single object or "shard"
   - shard = a horizontal partition of data in a database

Comparison from Cattell’s paper (2011)

System	Conc Contol	Data Storage	Repli-cation	Tx
Redis	Locks	RAM	Async	N
Scalaris	Locks	RAM	Sync	L
Tokyo	Locks	RAM or disk	Async	L
Voldemort	MVCC	RAM or BDB	Async	N
Riak	MVCC	Plug-in	Async	N
Membrain	Locks	Flash + Disk	Sync	L
Membase	Locks	Disk	Sync	L
Dynamo	MVCC	Plug-in	Async	N
SimpleDB	None	S3	Async	N
MongoDB	Locks	Disk	Async	N
Couch DB	MVCC	Disk	Async	N
Terrastore	Locks	RAM+	Sync	L
HBase	Locks	Hadoop	Async	L
HyperTable	Locks	Files	Sync	L
Cassandra	MVCC	Disk	Async	L
BigTable	Locks+stamps	GFS	Sync+ Async	L
PNUTs	MVCC	Disk	Async	L
MySQL Cluster	ACID	Disk	Sync	Y
VoltDB	ACID, no lock	RAM	Sync	Y
Clustrix	ACID, no lock	Disk	Sync	Y
ScaleDB	ACID	Disk	Sync	Y
ScaleBase	ACID	Disk	Async	Y
NimbusDB	ACID, no lock	Disk	Sync	Y

Data Store Categories

- The data stores are grouped according to their data model
- Key-value Stores:
  - store values and an index to find them based on a programmer-defined key
  - e.g. Project Voldemort, Riak, Redis, Scalaris, Tokyo Cabinet, Memcached/Membrain/Membase
- Document Stores:
  - store documents, which are indexed, with a simple query mechanism
  - e.g. Amazon SimpleDB, CouchDB, MongoDB, Terrastore
- Extensible Record Stores:
  - store extensible records that can be partitioned vertically and horizontally across nodes ("wide column stores")
  - e.g. Hbase, HyperTable, Cassandra, Yahoo's PNUTS
- Relational Databases:
  - store (and index and query) tuples, e.g. the new RDBMSs that provide horizontal scaling
  - e.g. MySQL Cluster, VoltDB, Clustrix, ScaleDB, ScaleBase, NimbusDB, Google Megastore (a layer on BigTable)

RDBMS benefits

- Relational DBMSs have taken and retained majority market share over other competitors in the past 30 years

- While no "one size fits all" in the SQL products themselves, there is a common interface with SQL, transactions, and relational schema that give advantages in training, continuity, and data interchange

- Successful relational DBMSs have been built to handle other specific application loads in the past:
  - read-only or read-mostly data warehousing, OLTP on multi-core multi-disk CPUs, in-memory databases, distributed databases, and now horizontally scaled databases

NoSQL benefits

- We haven't yet seen good benchmarks showing that RDBMSs can achieve scaling comparable with NoSQL systems like Google's BigTable
- If you only require a lookup of objects based on a single key, then a key-value/document store may be adequate and probably easier to understand than a relational DBMS
- Some applications require a flexible schema
- A relational DBMS makes "expensive" (multi- node multi-table) operations "too easy"
  - NoSQL systems make them impossible or obviously expensive for programmers
- The new systems are slowly gaining market shares too

Column Store

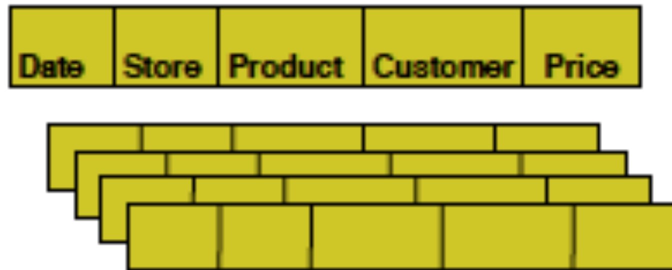
Row vs. Column Store

- Row store
  - store all attributes of a tuple together
  - storage like “row-major order” in a matrix
- Column store
  - store all rows for an attribute (column) together
  - storage like “column-major order” in a matrix
- e.g.
  - MonetDB, Vertica (earlier, C-store), SAP/Sybase IQ, Google Bigtable (with column groups)

Re-use permitted when acknowledging the original © Stavros Harizopoulos, Daniel Abadi, Peter Boncz (2009)

What is a column-store?

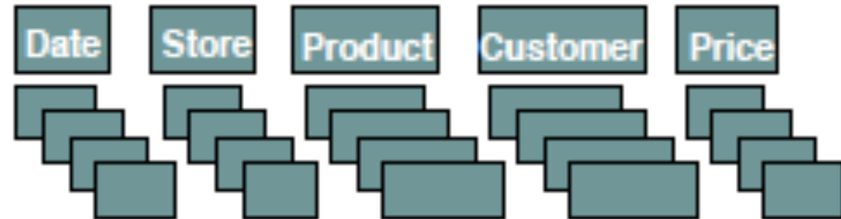
row-store



+ easy to add/modify a record

- might read in unnecessary data

column-store



+ only need to read in relevant data

- tuple writes require multiple accesses

=> suitable for read-mostly, read-intensive, large data repositories

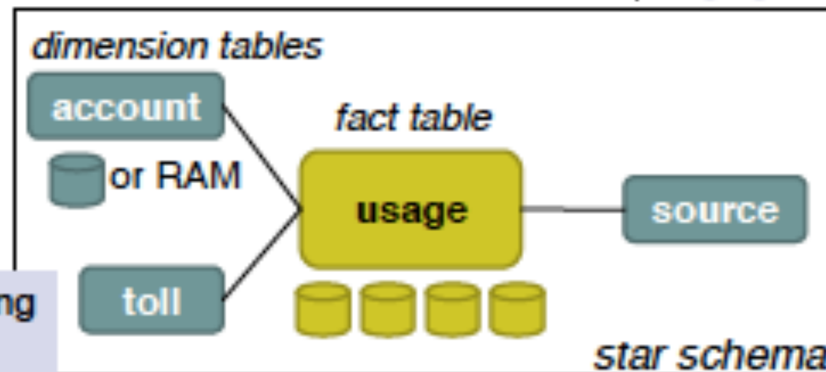
Ack: Slide from VLDB 2009 tutorial on Column store

Re-use permitted when acknowledging the original © Stavros Harizopoulos, Daniel Abadi, Peter Boncz (2009)

Telco Data Warehousing example

- Typical DW installation
- Real-world example

"One Size Fits All? - Part 2: Benchmarking Results" Stonebraker et al. CIDR 2007




```
QUERY 2
SELECT account.account_number,
sum (usage.toll_airtime),
sum (usage.toll_price)
FROM usage, toll, source, account
WHERE usage.toll_id = toll.toll_id
AND usage.source_id = source.source_id
AND usage.account_id = account.account_id
AND toll.type_ind in ('AE'. 'AA')
AND usage.toll_price > 0
AND source.type != 'CIBER'
AND toll.rating_method = 'IS'
AND usage.invoice_date = 20051013
GROUP BY account.account_number
```

	Column-store	Row-store
Query 1	2.06	300
Query 2	2.20	300
Query 3	0.09	300
Query 4	5.24	300
Query 5	2.88	300

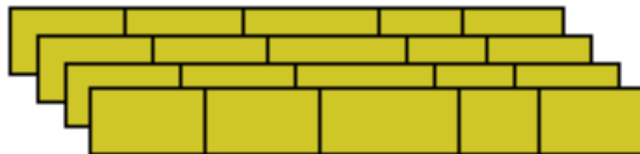
Why? Three main factors (next slides)

Ack: Slide from VLDB 2009 tutorial on Column store

Re-use permitted when acknowledging the original © Stavros Harizopoulos, Daniel Abadi, Peter Boncz (2009)

Telco example explained (1/3): *read efficiency*

row store



read pages containing entire rows

one row = 212 columns!

is this typical? (it depends)

What about vertical partitioning? (it does not work with ad-hoc queries)

column store

read only columns needed

in this example: 7 columns

caveats:

- "select * " not any faster
- clever disk prefetching
- clever tuple reconstruction

Ack: Slide from VLDB 2009 tutorial on Column store

Re-use permitted when acknowledging the original © Stavros Harizopoulos, Daniel Abadi, Peter Boncz (2009)

Telco example explained (2/3): *compression efficiency*

- Columns compress better than rows
  - Typical row-store compression ratio 1 : 3
  - Column-store 1 : 10

- Why?
  - Rows contain values from different domains
    => more entropy, difficult to dense-pack
  - Columns exhibit significantly less entropy
  - Examples:

Male, Female, Female, Female, Male
1998, 1998, 1999, 1999, 1999, 2000

  - Caveat: CPU cost (use lightweight compression)

Ack: Slide from VLDB 2009 tutorial on Column store

Re-use permitted when acknowledging the original © Stavros Harizopoulos, Daniel Abadi, Peter Boncz (2009)

Telco example explained (3/3): *sorting & indexing efficiency*

- Compression and dense-packing free up space
  - Use multiple overlapping column collections
  - Sorted columns compress better
  - Range queries are faster
  - Use sparse clustered indexes

Ack: Slide from VLDB 2009 tutorial on Column store