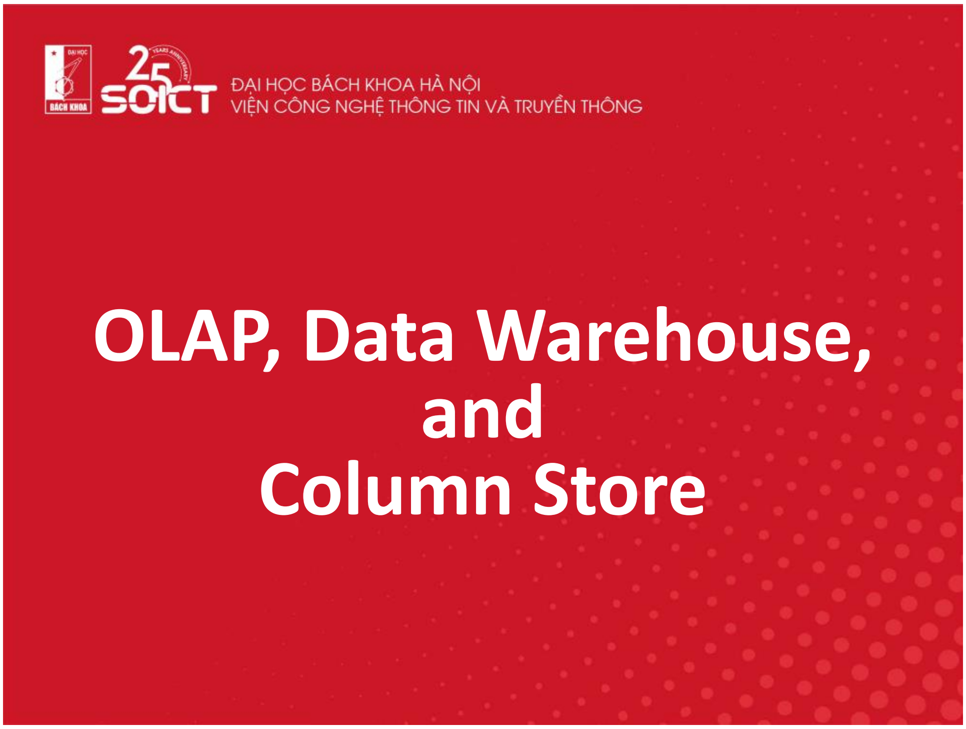
ĐẠI HỌC BÁCH KHOA HÀ NỘI
VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

1



ĐẠI HỌC BÁCH KHOA HÀ NỘI
VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

OLAP, Data Warehouse, and Column Store

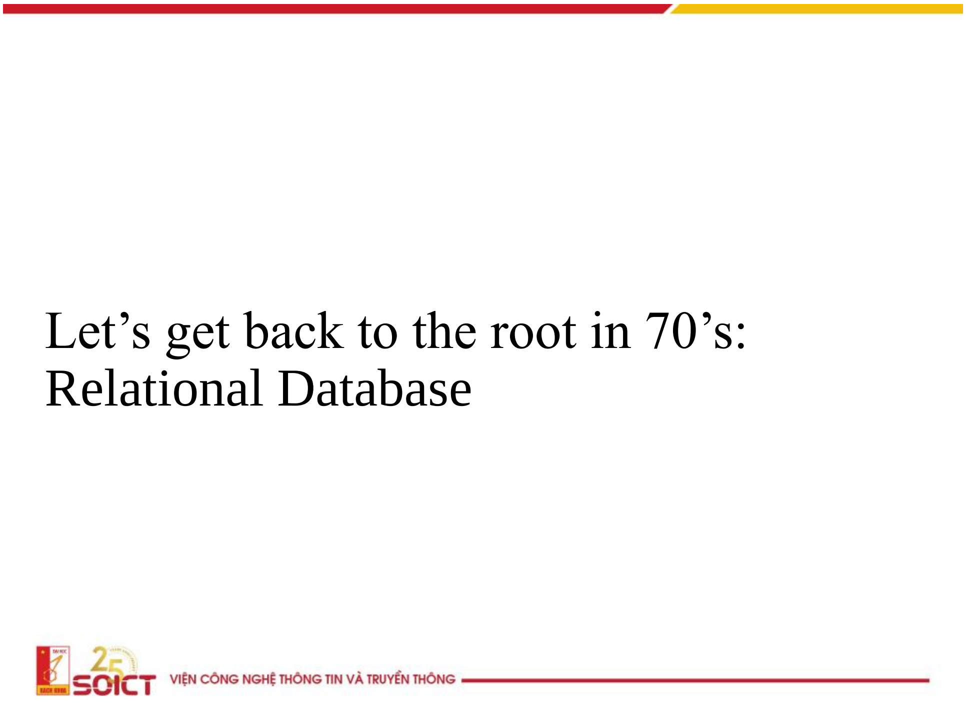
2

Why we still study OLAP/Data Warehouse in BI?

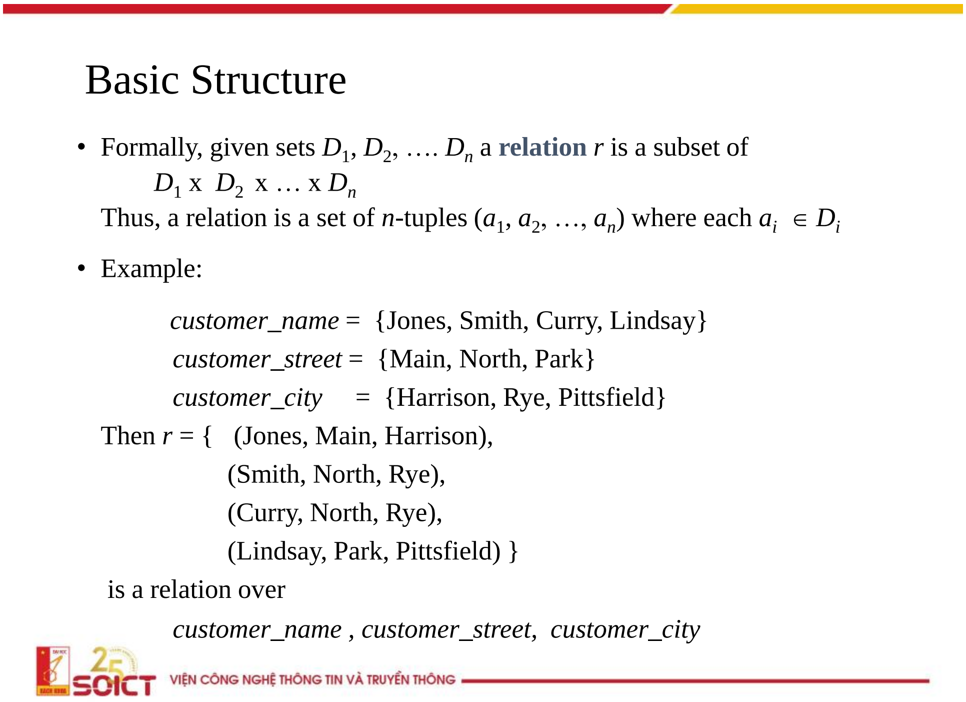
- Understand the Big Data history
  - How does the requirement of (big) data analytics/business intelligence evolve over the time?
  - What are the architecture and implementation techniques being developed? Will they still be useful in Big Data?
  - Understand their limitation and what factors have changed from 90’s to now?
- NoSQL is not only SQL☺
- Hive/Impala aims to provide OLAP/BI for Big Data using Hadoop

Highlights

- OLAP
  - Multi-relational Data model
  - Operators
  - SQL
- Data warehouse (architecture, issues, optimizations)
- Join Processing
- Column Stores (Optimized for OLAP workload)



Let’s get back to the root in 70’s: Relational Database



Basic Structure

- Formally, given sets D_1, D_2, …. D_n a **relation** r is a subset of
  $D_1 \times D_2 \times \ldots \times D_n$
  Thus, a relation is a set of n-tuples $(a_1, a_2, \ldots, a_n)$ where each $a_i \in D_i$
- Example:

  customer_name = {Jones, Smith, Curry, Lindsay}
  customer_street = {Main, North, Park}
  customer_city = {Harrison, Rye, Pittsfield}

  Then r = { (Jones, Main, Harrison),
  (Smith, North, Rye),
  (Curry, North, Rye),
  (Lindsay, Park, Pittsfield) }

  is a relation over

  customer_name , *customer_street*, *customer_city*

Relation Schema

- $A_1, A_2, \ldots, A_n$ are *attributes*
- $R = (A_1, A_2, \ldots, A_n)$ is a *relation schema*

  Example:

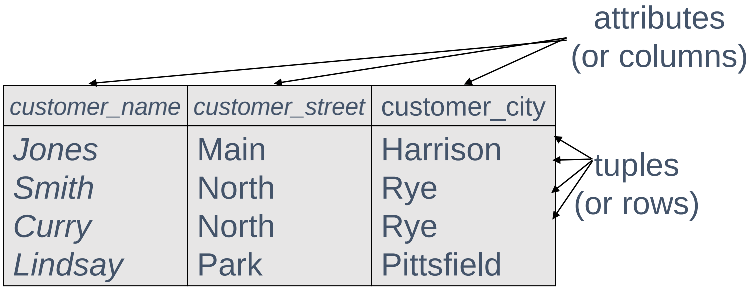
  Customer_schema = (*customer_name*, *customer_street*, *customer_city*)
- *r*(*R*) is a *relation* on the *relation schema R*

  Example:

  customer (Customer_schema)

Relation Instance

- The current values (*relation instance*) of a relation are specified by a table
- An element *t* of *r* is a *tuple*, represented by a *row* in a table



customer_name	*customer_street*	customer_city
Jones	Main	Harrison
Smith	North	Rye
Curry	North	Rye
Lindsay	Park	Pittsfield

customer

Database

- A database consists of multiple relations
- Information about an enterprise is broken up into parts, with each relation storing one part of the information

account : stores information about accounts
depositor : stores information about which customer owns which account
customer : stores information about customers

- Storing all information as a single relation such as
  bank(*account_number, balance, customer_name*, ..)
  results in repetition of information (e.g., two customers own an account) and the need for null values (e.g., represent a customer without an account)

Banking Example

branch (branch-name, branch-city, assets)

customer (customer-name, customer-street, customer-city)

account (account-number, branch-name, balance)

loan (loan-number, branch-name, amount)

depositor (customer-name, account-number)

borrower (customer-name, loan-number)

Relational Algebra

- Primitives
  - Projection (π)
  - Selection (σ)
  - Cartesian product (×)
  - Set union (∪)
  - Set difference (−)
  - Rename (ρ)
- Other operations
  - Join (⋈)
  - Group by… aggregation
  - …

What happens next?

- SQL
- System R (DB2), INGRES, ORACLE, SQL-Server, Teradata
  - –B+-Tree (select)
  - –Transaction Management
  - –Join algorithm

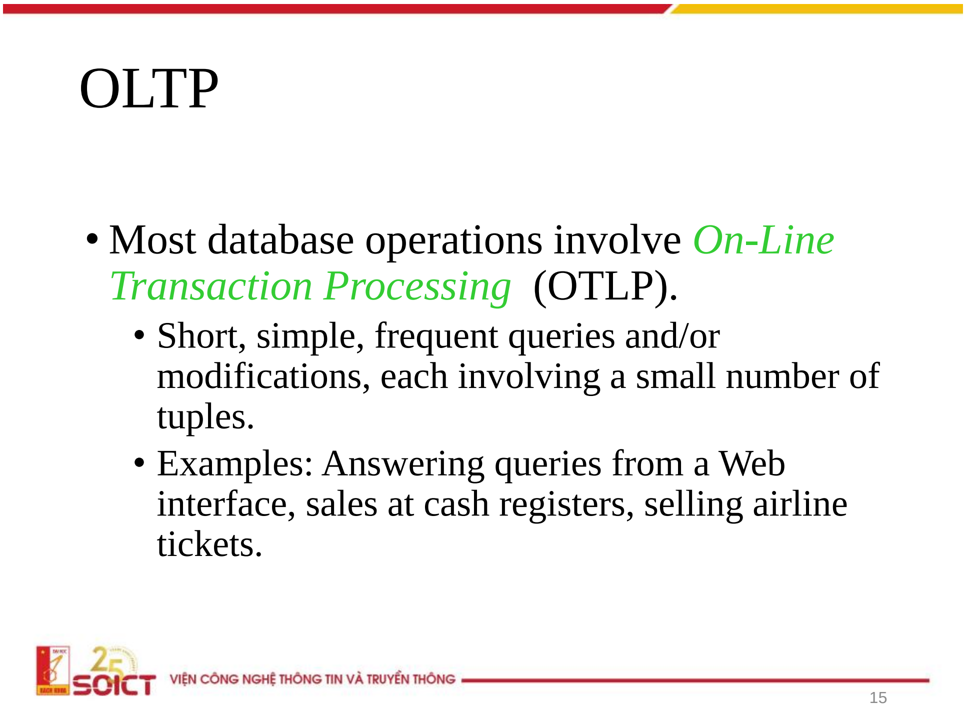
In early 90's:
OLAP & Data Warehouse

13

Database Workloads

- OLTP (online transaction processing)
  - Typical applications: e-commerce, banking, airline reservations
  - User facing: real-time, low latency, highly-concurrent
  - Tasks: relatively small set of "standard" transactional queries
  - Data access pattern: random reads, updates, writes (involving relatively small amounts of data)
- OLAP (online analytical processing)
  - Typical applications: business intelligence, data mining
  - Back-end processing: batch workloads, less concurrency
  - Tasks: complex analytical queries, often ad hoc
  - Data access pattern: table scans, large amounts of data involved per query

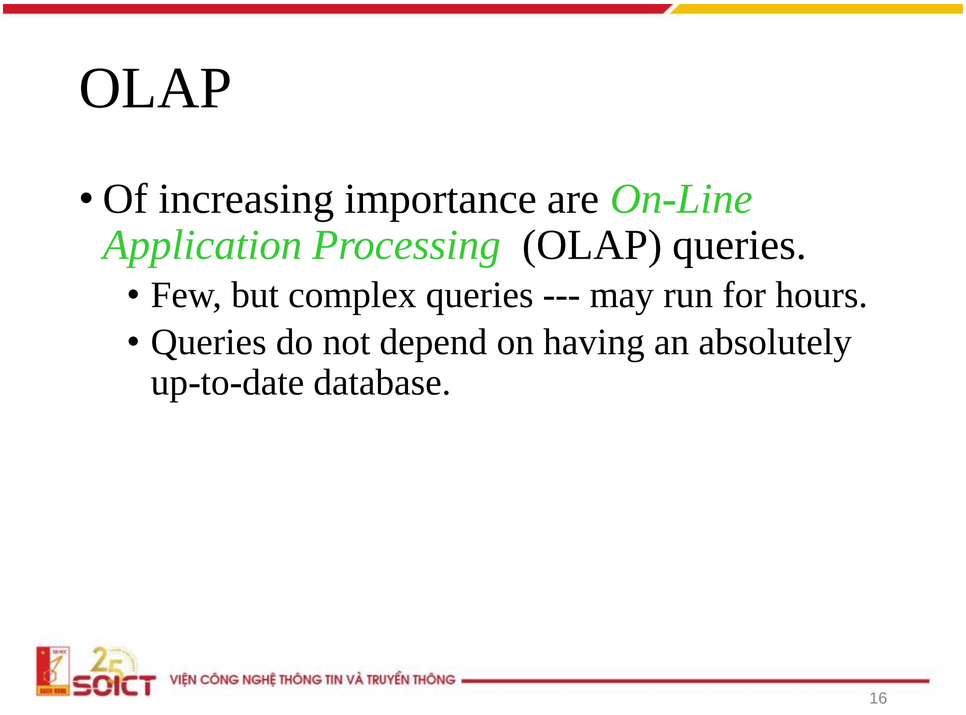
14



OLTP

- Most database operations involve *On-Line Transaction Processing* (OTLP).
  - Short, simple, frequent queries and/or modifications, each involving a small number of tuples.
  - Examples: Answering queries from a Web interface, sales at cash registers, selling airline tickets.

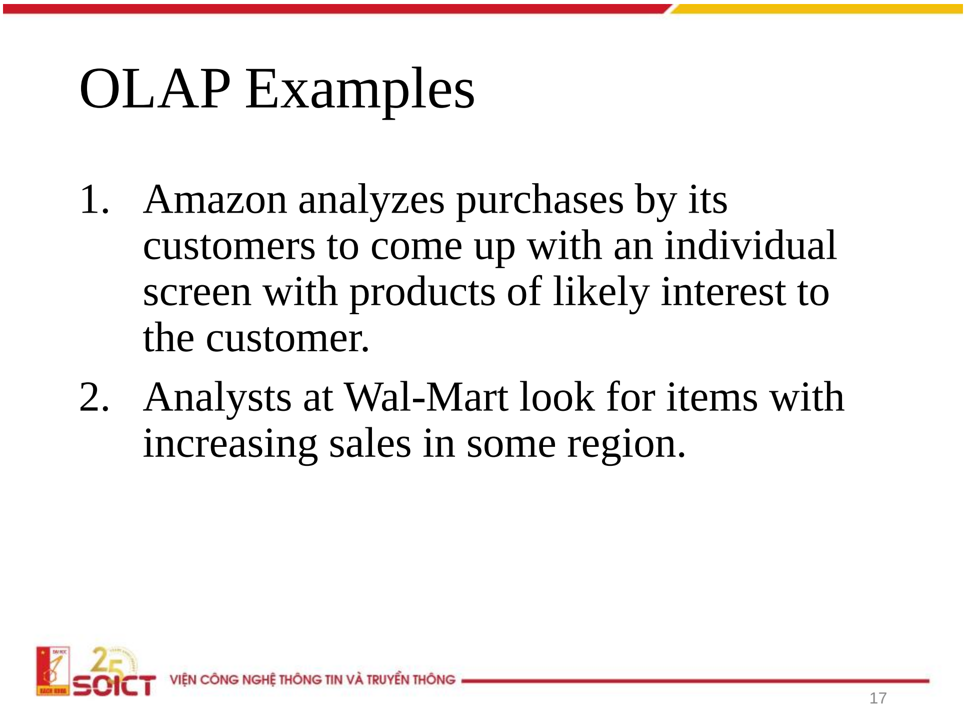
15



OLAP

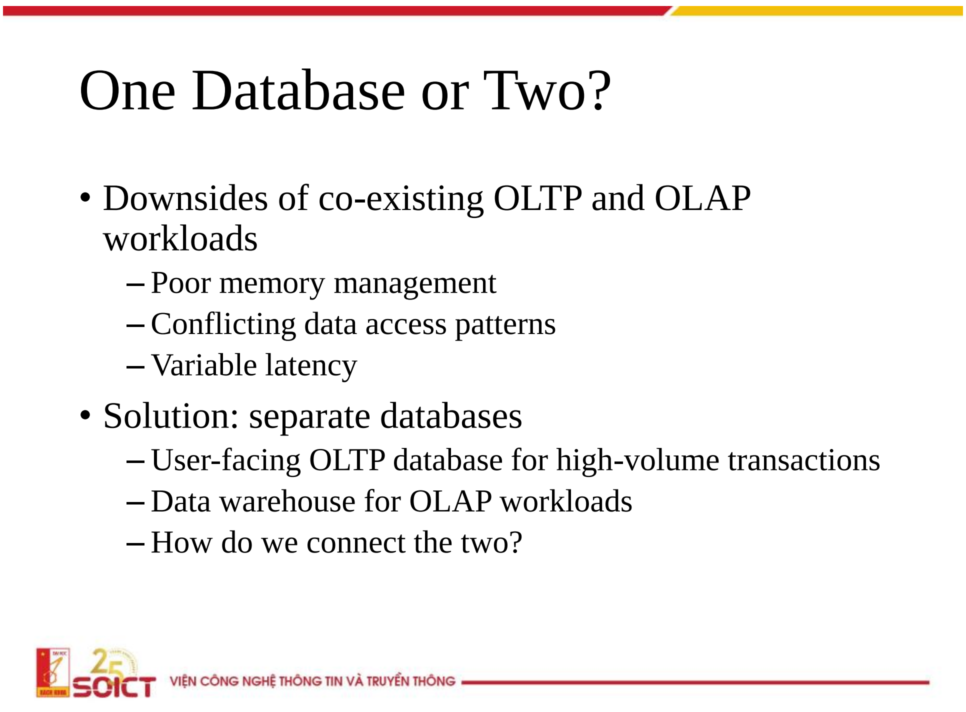
- Of increasing importance are *On-Line Application Processing* (OLAP) queries.
  - Few, but complex queries --- may run for hours.
  - Queries do not depend on having an absolutely up-to-date database.

16



OLAP Examples

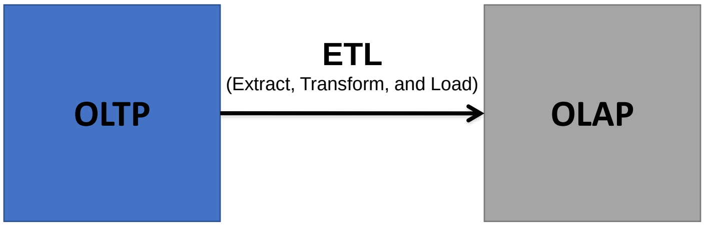
1. Amazon analyzes purchases by its customers to come up with an individual screen with products of likely interest to the customer.
2. Analysts at Wal-Mart look for items with increasing sales in some region.



One Database or Two?

- Downsides of co-existing OLTP and OLAP workloads
  - Poor memory management
  - Conflicting data access patterns
  - Variable latency
- Solution: separate databases
  - User-facing OLTP database for high-volume transactions
  - Data warehouse for OLAP workloads
  - How do we connect the two?

OLTP/OLAP Architecture



OLTP

ETL

(Extract, Transform, and Load)

OLAP

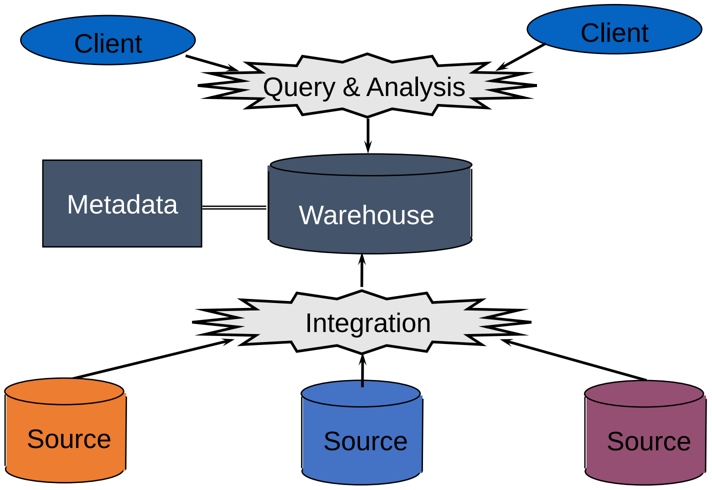
OLTP/OLAP Integration

- OLTP database for user-facing transactions
  - Retain records of all activity
  - Periodic ETL (e.g., nightly)
- Extract-Transform-Load (ETL)
  - Extract records from source
  - Transform: clean data, check integrity, aggregate, etc.
  - Load into OLAP database
- OLAP database for data warehousing
  - Business intelligence: reporting, ad hoc queries, data mining, etc.
  - Feedback to improve OLTP services

The Data Warehouse

- The most common form of data integration.
  - Copy sources into a single DB (*warehouse*) and try to keep it up-to-date.
  - Usual method: periodic reconstruction of the warehouse, perhaps overnight.
  - Frequently essential for analytic queries.

Warehouse Architecture



Star Schemas

- A *star schema* is a common organization for data at a warehouse. It consists of:
  1. *Fact table* : a very large accumulation of facts such as sales.
     - Often “insert-only.”
  2. *Dimension tables* : smaller, generally static information about the entities involved in the facts.

Example: Star Schema

- Suppose we want to record in a warehouse information about every beer sale: the bar, the brand of beer, the drinker who bought the beer, the day, the time, and the price charged.
- The fact table is a relation:

Sales(bar, beer, drinker, day, time, price)

Example, Continued

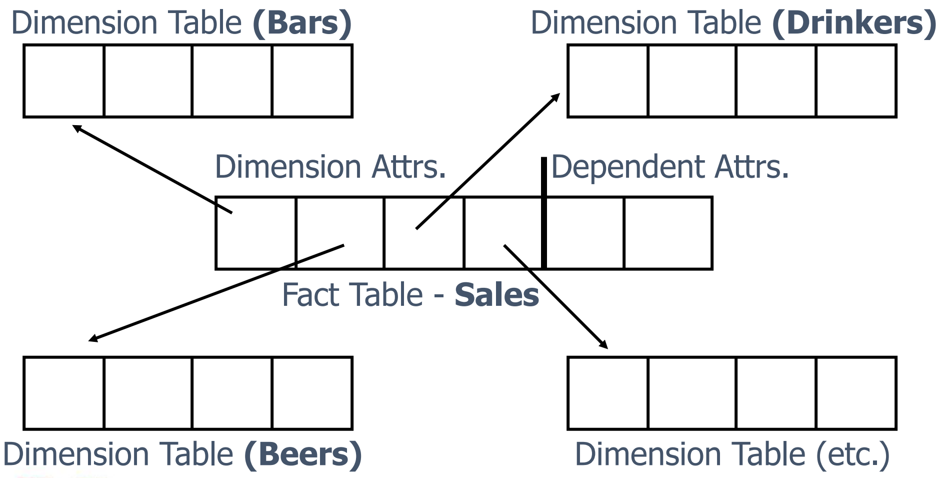
- The dimension tables include information about the bar, beer, and drinker "dimensions":

  Bars(bar, addr, license)

  Beers(beer, manf)

  Drinkers(drinker, addr, phone)

Visualization – Star Schema



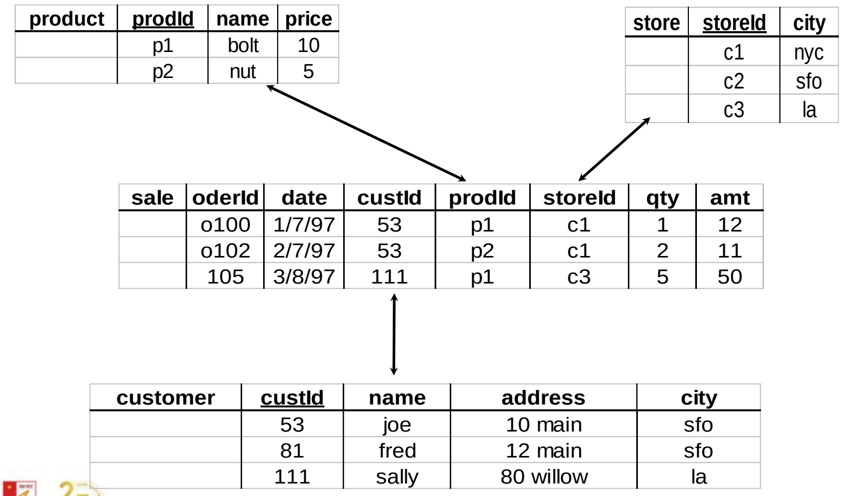
Dimensions and Dependent Attributes

- Two classes of fact-table attributes:
  1. *Dimension attributes* : the key of a dimension table.
  2. *Dependent attributes* : a value determined by the dimension attributes of the tuple.

Warehouse Models & Operators

- Data Models
  - relations
  - stars & snowflakes
  - cubes
- Operators
  - slice & dice
  - roll-up, drill down
  - pivoting
  - other

Star



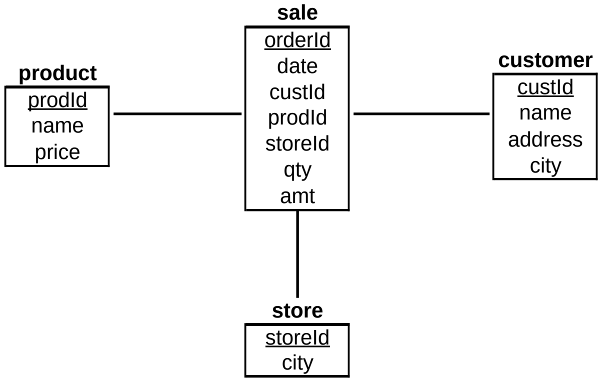
product	prodId	name	price
	p1	bolt	10
	p2	nut	5

store	storeId	city
	c1	nyc
	c2	sfo
	c3	la

sale	oderId	date	custId	prodId	storeId	qty	amt
	o100	1/7/97	53	p1	c1	1	12
	o102	2/7/97	53	p2	c1	2	11
	105	3/8/97	111	p1	c3	5	50

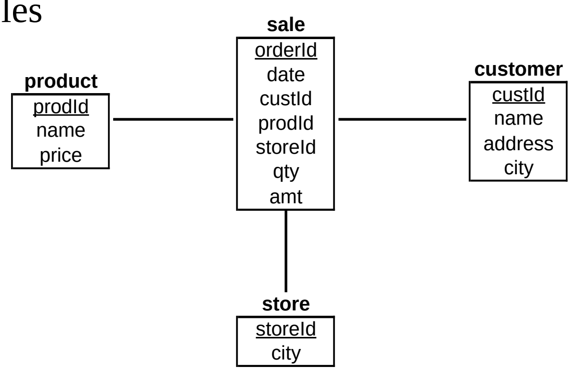
customer	custId	name	address	city
	53	joe	10 main	sfo
	81	fred	12 main	sfo
	111	sally	80 willow	la

Star Schema



Terms

- Fact table
- Dimension tables
- Measures

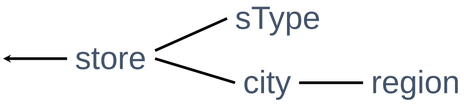

sale
orderId
date
custId
prodId
storeId
qty
amt

product
prodId
name
price

customer
custId
name
address
city

store
storeId
city

Dimension Hierarchies



store	storeId	cityId	tId	mgr
	s5	sfo	t1	joe
	s7	sfo	t2	fred
	s9	la	t1	nancy

sType	tId	size	location
	t1	small	downtown
	t2	large	suburbs

city	cityId	pop	regId
	sfo	1M	north
	la	5M	south

region	regId	name
	north	cold region
	south	warm region

➔ snowflake schema
➔ constellations

Aggregates

- Add up amounts for day 1
- In SQL: SELECT sum(amt) FROM SALE
  WHERE date = 1

sale	prodId	storeId	date	amt
	p1	c1	1	12
	p2	c1	1	11
	p1	c3	1	50
	p2	c2	1	8
	p1	c1	2	44
	p1	c2	2	4

➡ 81

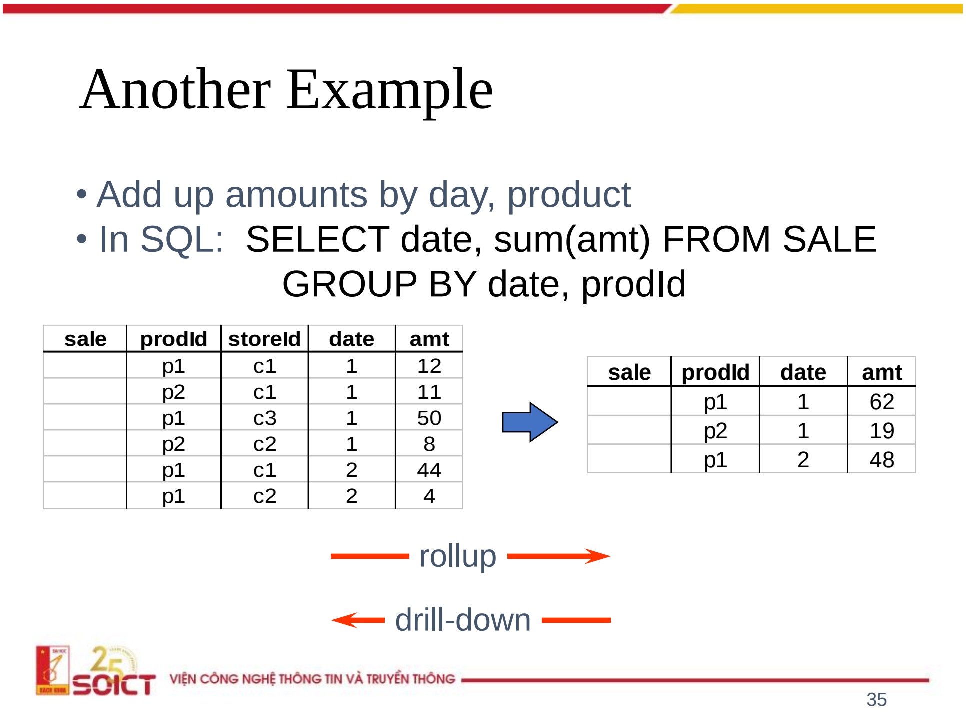
Aggregates

- Add up amounts by day
- In SQL: SELECT date, sum(amt) FROM SALE
  GROUP BY date

sale	prodId	storeId	date	amt
	p1	c1	1	12
	p2	c1	1	11
	p1	c3	1	50
	p2	c2	1	8
	p1	c1	2	44
	p1	c2	2	4

➡

ans	date	sum
	1	81
	2	48



Another Example

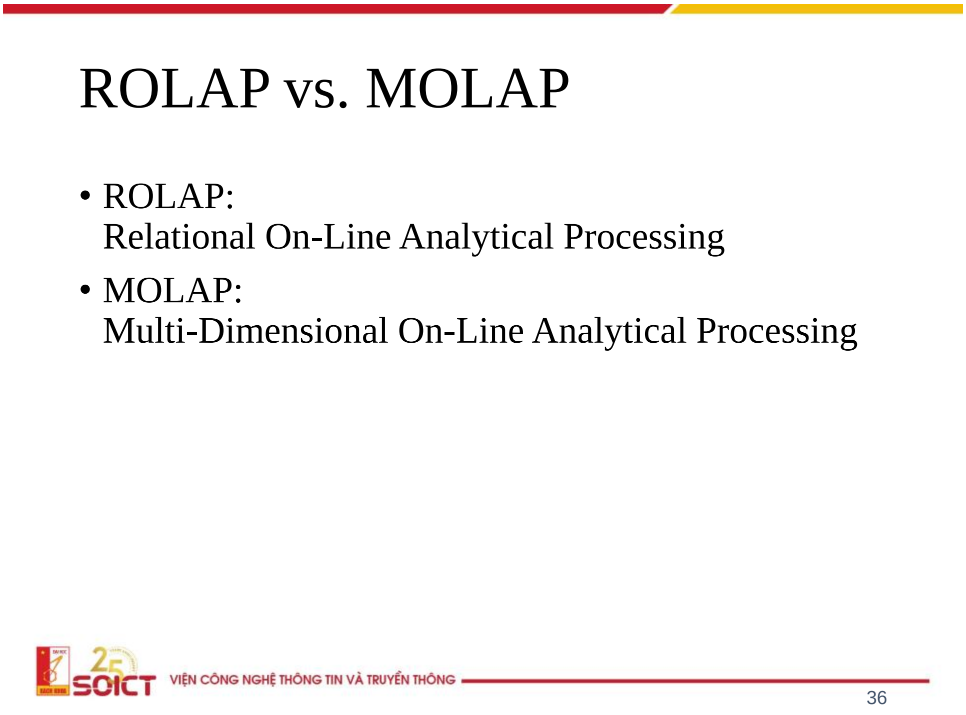
- Add up amounts by day, product
- In SQL: SELECT date, sum(amt) FROM SALE GROUP BY date, prodId

sale	prodId	storeId	date	amt
	p1	c1	1	12
	p2	c1	1	11
	p1	c3	1	50
	p2	c2	1	8
	p1	c1	2	44
	p1	c2	2	4

sale	prodId	date	amt
	p1	1	62
	p2	1	19
	p1	2	48

rollup →

← drill-down



ROLAP vs. MOLAP

- ROLAP:
  Relational On-Line Analytical Processing
- MOLAP:
  Multi-Dimensional On-Line Analytical Processing

Cube

Fact table view:

sale	prodId	storeId	amt
	p1	c1	12
	p2	c1	11
	p1	c3	50
	p2	c2	8

↔

Multi-dimensional cube:

	c1	c2	c3
p1	12		50
p2	11	8	

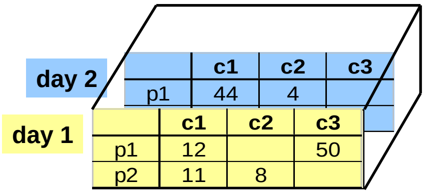
dimensions = 2

3-D Cube

Fact table view:

sale	prodId	storeId	date	amt
	p1	c1	1	12
	p2	c1	1	11
	p1	c3	1	50
	p2	c2	1	8
	p1	c1	2	44
	p1	c2	2	4

Multi-dimensional cube:



day 2

	c1	c2	c3
p1	44	4	

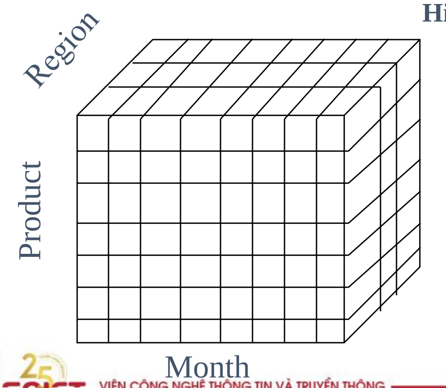
day 1

	c1	c2	c3
p1	12		50
p2	11	8	

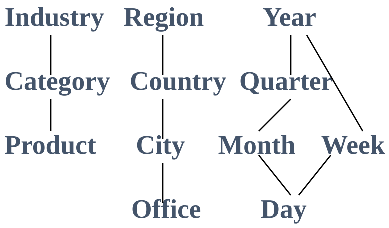
dimensions = 3

Multidimensional Data

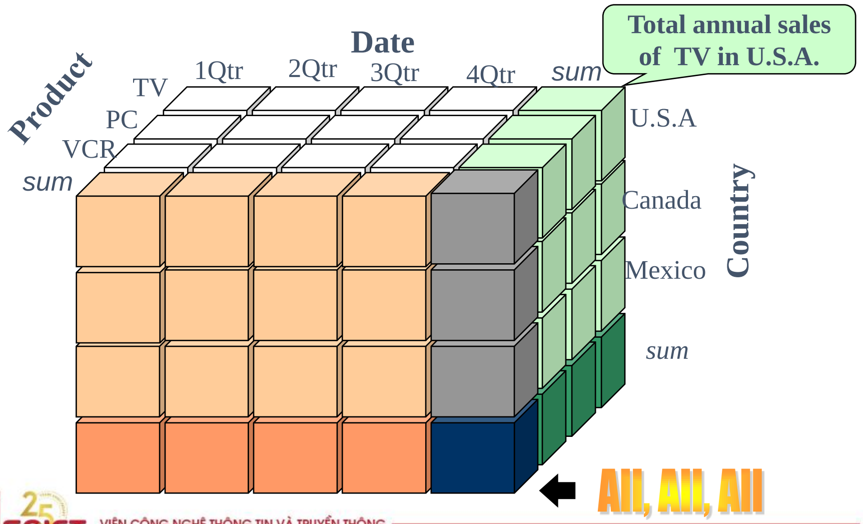
- Sales volume as a function of product, month, and region



Dimensions: Product, Location, Time
Hierarchical summarization paths

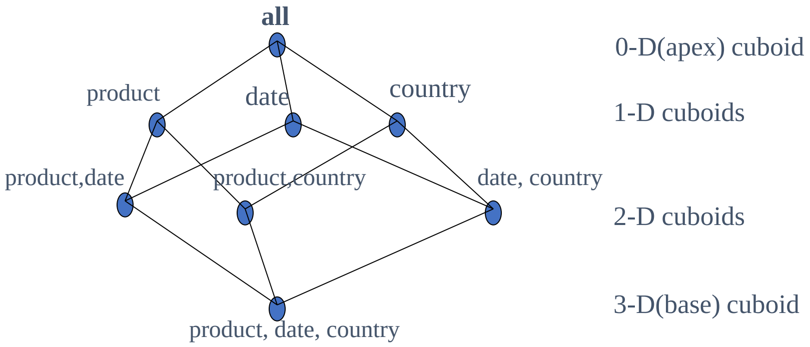


A Sample Data Cube

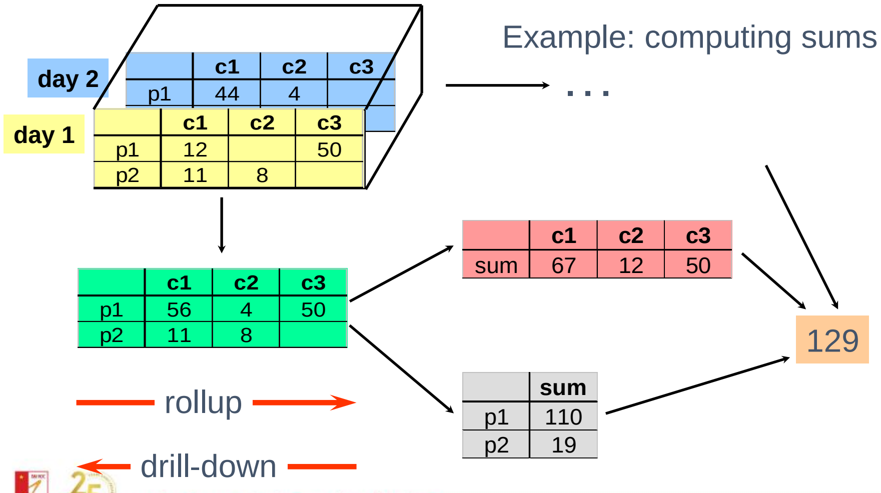


Total annual sales of TV in U.S.A.

Cuboids Corresponding to the Cube



Cube Aggregation



day 2

	c1	c2	c3
p1	44	4	

day 1

	c1	c2	c3
p1	12		50
p2	11	8	

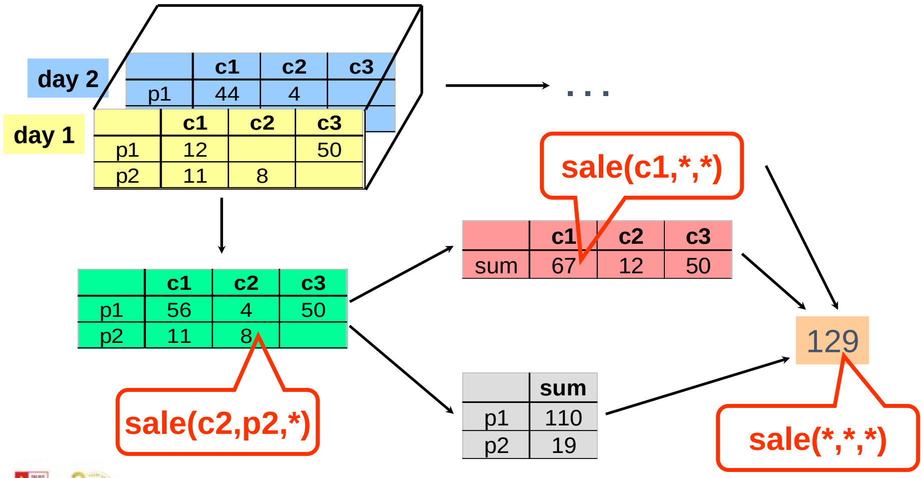
	c1	c2	c3
p1	56	4	50
p2	11	8	

	c1	c2	c3
sum	67	12	50

	sum
p1	110
p2	19

129

Cube Operators



day 2

	c1	c2	c3
p1	44	4	

day 1

	c1	c2	c3
p1	12		50
p2	11	8	

. . .

	c1	c2	c3
p1	56	4	50
p2	11	8	

sale(c2,p2,*)

	c1	c2	c3
sum	67	12	50

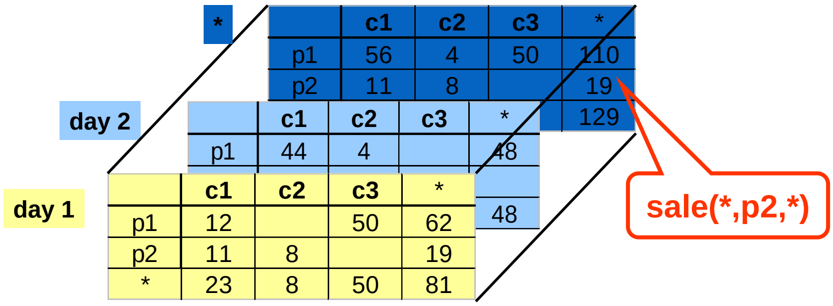
sale(c1,*,*)

	sum
p1	110
p2	19

129

sale(*,*,*)

Extended Cube



*

	c1	c2	c3	*
p1	56	4	50	110
p2	11	8		19
				129

day 2

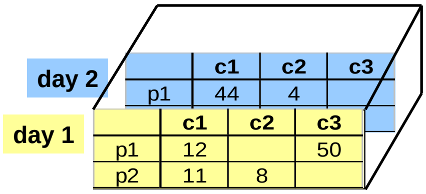
	c1	c2	c3	*
p1	44	4		48
				48

day 1

	c1	c2	c3	*
p1	12		50	62
p2	11	8		19
*	23	8	50	81

sale(*,p2,*)

Aggregation Using Hierarchies



day 2

	c1	c2	c3
p1	44	4	

day 1

	c1	c2	c3
p1	12		50
p2	11	8	

	region A	region B
p1	56	54
p2	11	8

customer
|
region
|
country

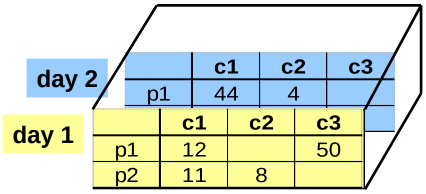
(customer c1 in Region A;
customers c2, c3 in Region B)

Pivoting

Fact table view:

sale	prodId	storeId	date	amt
	p1	c1	1	12
	p2	c1	1	11
	p1	c3	1	50
	p2	c2	1	8
	p1	c1	2	44
	p1	c2	2	4

Multi-dimensional cube:



day 2

	c1	c2	c3
p1	44	4	

day 1

	c1	c2	c3
p1	12		50
p2	11	8	

	c1	c2	c3
p1	56	4	50
p2	11	8	

CUBE Operator (SQL-99)

Chevy Sales Cross Tab				
Chevy	*1990*	*1991*	*1992*	*Total (ALL)*
black	50	85	154	289
white	40	115	199	354
Total (ALL)	90	200	353	1286

```
SELECT   model, year, color, sum(sales) as sales
FROM     sales
WHERE    model in (‘Chevy’)
AND      year BETWEEN 1990 AND 1992
GROUP BY   CUBE (model, year, color);
```

CUBE Contd.

```
SELECT   model, year, color, sum(sales) as sales
FROM     sales
WHERE    model in (‘Chevy’)
AND      year BETWEEN 1990 AND 1992
GROUP BY       CUBE (model, year, color);
```

- Computes union of 8 different groupings:
  - {(model, year, color), (model, year), (model, color), (year, color), (model), (year), (color), ()}

Aggregates

- Operators: sum, count, max, min, median, average
- “Having” clause
- Cube (& Rollup) operator
- Using dimension hierarchy
  - average by region (within store)
  - maximum by month (within date)

Query & Analysis Tools

- Query Building
- Report Writers (comparisons, growth, graphs,…)
- Spreadsheet Systems
- Web Interfaces
- Data Mining

Other Operations

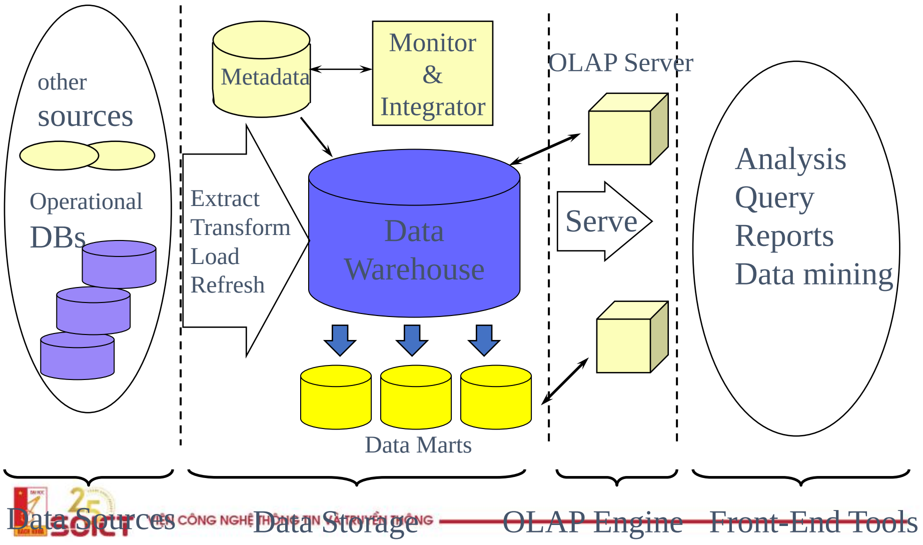
- Time functions
  - e.g., time average
- Computed Attributes
  - e.g., commission = sales * rate
- Text Queries
  - e.g., find documents with words X AND B
  - e.g., rank documents by frequency of words X, Y, Z

Data Warehouse Implementation

Implementing a Warehouse

- *Monitoring*: Sending data from sources
- *Integrating*: Loading, cleansing,...
- *Processing*: Query processing, indexing, ...
- *Managing*: Metadata, Design, ...

Multi-Tiered Architecture



Monitoring

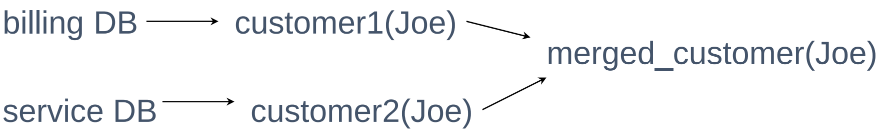
- Source Types: relational, flat file, IMS, VSAM, IDMS, WWW, news-wire, …
- Incremental vs. Refresh

customer	id	name	address	city
	53	joe	10 main	sfo
	81	fred	12 main	sfo
	111	sally	80 willow	la

new

Data Cleaning

- Migration (e.g., yen ⇨ dollars)
- Scrubbing: use domain-specific knowledge (e.g., social security numbers)
- Fusion (e.g., mail list, customer merging)
- Auditing: discover rules & relationships (like data mining)



billing DB → customer1(Joe) → merged_customer(Joe)

service DB → customer2(Joe) → merged_customer(Joe)

Loading Data

- Incremental vs. refresh
- Off-line vs. on-line
- Frequency of loading
  - At night, 1x a week/month, continuously
- Parallel/Partitioned load

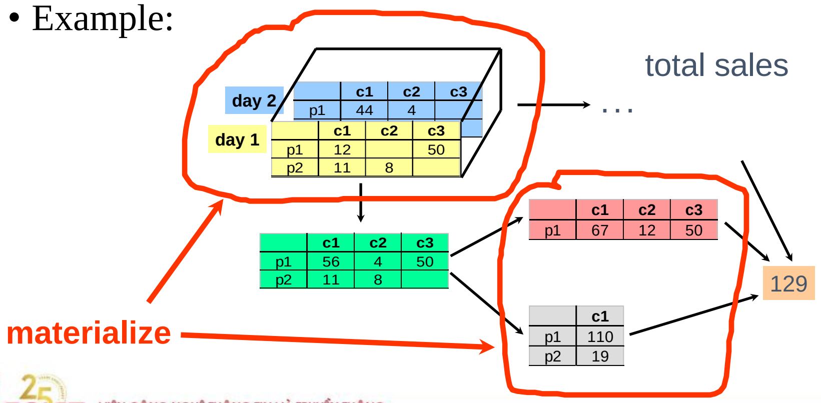
OLAP Implementation

Derived Data

- Derived Warehouse Data
  - indexes
  - aggregates
  - materialized views (next slide)
- When to update derived data?
- Incremental vs. refresh

What to Materialize?

- Store in warehouse results useful for common queries
- Example:



day 2

	c1	c2	c3
p1	44	4	

day 1

	c1	c2	c3
p1	12		50
p2	11	8	

	c1	c2	c3
p1	56	4	50
p2	11	8	

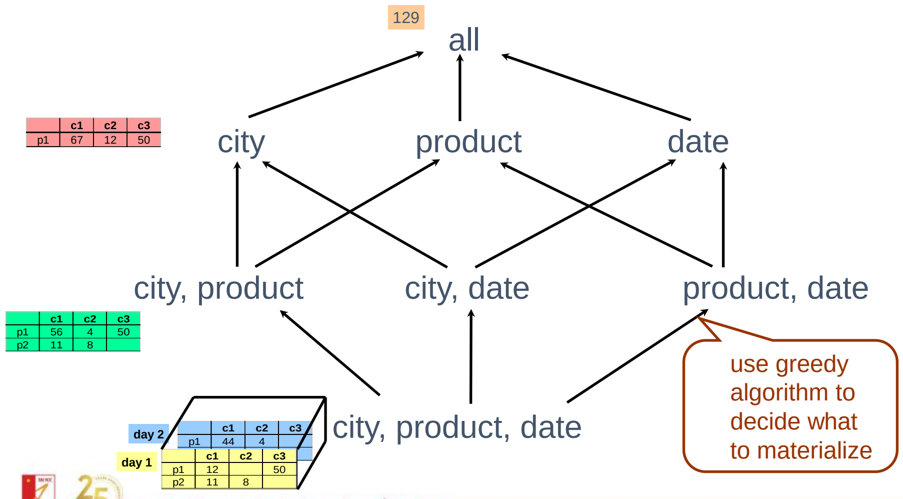
	c1	c2	c3
p1	67	12	50

	c1
p1	110
p2	19

Materialization Factors

- Type/frequency of queries
- Query response time
- Storage cost
- Update cost

Cube Aggregates Lattice



	c1	c2	c3
p1	67	12	50

	c1	c2	c3
p1	56	4	50
p2	11	8	

day 2

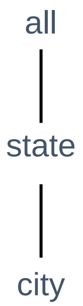
	c1	c2	c3
p1	44	4	

day 1

	c1	c2	c3
p1	12		50
p2	11	8	

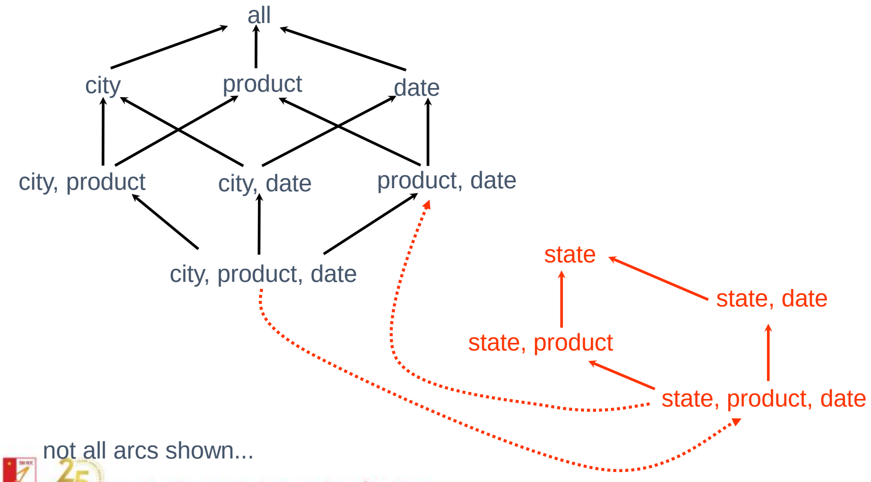
use greedy algorithm to decide what to materialize

Dimension Hierarchies



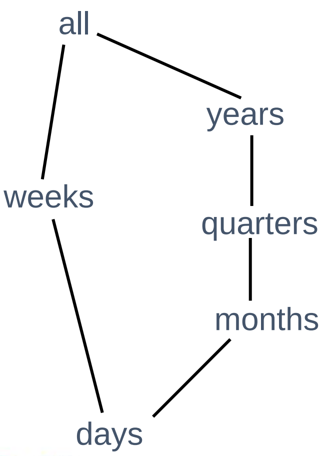
cities	city	state
	c1	CA
	c2	NY

Dimension Hierarchies



not all arcs shown...

Interesting Hierarchy



time	day	week	month	quarter	year
	1	1	1	1	2000
	2	1	1	1	2000
	3	1	1	1	2000
	4	1	1	1	2000
	5	1	1	1	2000
	6	1	1	1	2000
	7	1	1	1	2000
	8	2	1	1	2000

conceptual dimension table

Indexing OLAP Data: Bitmap Index

- Index on a particular column
- Each value in the column has a bit vector: bit-op is fast
- The length of the bit vector: # of records in the base table
- The *i*-th bit is set if the *i*-th row of the base table has the value for the indexed column
- not suitable for high cardinality domains

Base table

Cust	Region	Type
C1	Asia	Retail
C2	Europe	Dealer
C3	Asia	Dealer
C4	America	Retail
C5	Europe	Dealer

Index on Region

RecID	Asia	Europe	America
1	1	0	0
2	0	1	0
3	1	0	0
4	0	0	1
5	0	1	0

Index on Type

RecID	Retail	Dealer
1	1	0
2	0	1
3	0	1
4	1	0
5	0	1

Join Processing

Join

- How does DBMS join two tables?
- Sorting is one way...
- Database must choose best way for each query

Schema for Examples

Sailors (*sid*: integer, *sname*: string, *rating*: integer, *age*: real)
Reserves (*sid*: integer, *bid*: integer, *day*: dates, *rname*: string)

- Similar to old schema; *rname* added for variations.
- Reserves:
  - Each tuple is 40 bytes long,
  - 100 tuples per page,
  - M = 1000 pages total.
- Sailors:
  - Each tuple is 50 bytes long,
  - 80 tuples per page,
  - N = 500 pages total.

Equality Joins With One Join Column

```
SELECT  *
FROM    Reserves R1, Sailors S1
WHERE   R1.sid=S1.sid
```

- In algebra: $R \bowtie S$. Common! Must be carefully optimized. R × S is large; so, R × S followed by a selection is inefficient.
- Assume: M tuples in R, p_R tuples per page, N tuples in S, p_S tuples per page.
  - In our examples, R is Reserves and S is Sailors.
- We will consider more complex join conditions later.
- *Cost metric*: # of I/Os. We will ignore output costs.

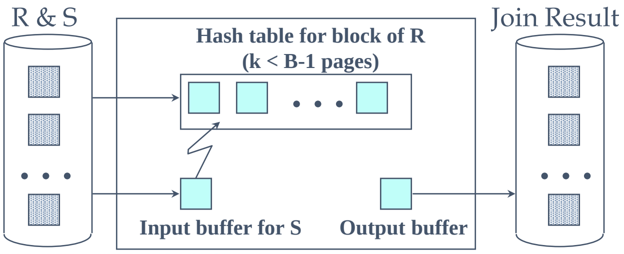
Simple Nested Loops Join

```
foreach tuple r in R do
    foreach tuple s in S do
        if $r_i$ == $s_j$ then add <r, s> to result
```

- For each tuple in the *outer* relation R, we scan the entire *inner* relation S.
  - Cost: M + p_R * M * N = 1000 + 100*1000*500 I/Os.
- Page-oriented Nested Loops join: For each *page* of R, get each *page* of S, and write out matching pairs of tuples <r, s>, where r is in R-page and S is in S-page.
  - Cost: M + M*N = 1000 + 1000*500
  - If smaller relation (S) is outer, cost = 500 + 500*1000

Block Nested Loops Join

- Use one page as an input buffer for scanning the inner S, one page as the output buffer, and use all remaining pages to hold ``block'' of outer R.
  - For each matching tuple r in R-block, s in S-page, add <r, s> to result. Then read next R-block, scan S, etc.



Examples of Block Nested Loops

- Cost: Scan of outer + #outer blocks * scan of inner
  - #outer blocks = $\lceil \#\ of\ \ pages\ of\ \ outer\ /\ blocksize \rceil$
- With Reserves (R) as outer, and 100 pages of R:
  - Cost of scanning R is 1000 I/Os; a total of 10 *blocks*.
  - Per block of R, we scan Sailors (S); 10*500 I/Os.
  - If space for just 90 pages of R, we would scan S 12 times.
- With 100-page block of Sailors as outer:
  - Cost of scanning S is 500 I/Os; a total of 5 blocks.
  - Per block of S, we scan Reserves; 5*1000 I/Os.
- With *sequential reads* considered, analysis changes: may be best to divide buffers evenly between R and S.

VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

Index Nested Loops Join

```
foreach tuple r in R do
    foreach tuple s in S where $r_i == s_j$ do
        add <r, s> to result
```

- If there is an index on the join column of one relation (say S), can make it the inner and exploit the index.
  - Cost: M + ((M*p_R) * cost of finding matching S tuples)
- For each R tuple, cost of probing S index is about 1.2 for hash index, 2-4 for B+ tree. Cost of then finding S tuples (assuming Alt. (2) or (3) for data entries) depends on clustering.
  - Clustered index: 1 I/O (typical), unclustered: upto 1 I/O per matching S tuple.

VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

Examples of Index Nested Loops

- Hash-index (Alt. 2) on *sid* of Sailors (as inner):
  - Scan Reserves: 1000 page I/Os, 100*1000 tuples.
  - For each Reserves tuple: 1.2 I/Os to get data entry in index, plus 1 I/O to get (the exactly one) matching Sailors tuple. Total: 220,000 I/Os.
- Hash-index (Alt. 2) on *sid* of Reserves (as inner):
  - Scan Sailors: 500 page I/Os, 80*500 tuples.
  - For each Sailors tuple: 1.2 I/Os to find index page with data entries, plus cost of retrieving matching Reserves tuples. Assuming uniform distribution, 2.5 reservations per sailor (100,000 / 40,000). Cost of retrieving them is 1 or 2.5 I/Os depending on whether the index is clustered.

Sort-Merge Join $(R \bowtie_{i=j} S)$

- Sort R and S on the join column, then scan them to do a ``merge'' (on join col.), and output result tuples.
  - Advance scan of R until current R-tuple >= current S tuple, then advance scan of S until current S-tuple >= current R tuple; do this until current R tuple = current S tuple.
  - At this point, all R tuples with same value in Ri (*current R group*) and all S tuples with same value in Sj (*current S group*) *match*; output <r, s> for all pairs of such tuples.
  - Then resume scanning R and S.
- R is scanned once; each S group is scanned once per matching R tuple. (Multiple scans of an S group are likely to find needed pages in buffer.)

Example of Sort-Merge Join

sid	sname	rating	age
22	dustin	7	45.0
28	yuppy	9	35.0
31	lubber	8	55.5
44	guppy	5	35.0
58	rusty	10	35.0

sid	bid	day	rname
28	103	12/4/96	guppy
28	103	11/3/96	yuppy
31	101	10/10/96	dustin
31	102	10/12/96	lubber
31	101	10/11/96	lubber
58	103	11/12/96	dustin

- Cost: M log M + N log N + (M+N)
  - The cost of scanning, M+N, could be M*N (very unlikely!)
- With 35, 100 or 300 buffer pages, both Reserves and Sailors can be sorted in 2 passes; total join cost: 7500.

(BNL cost: 2500 to 15000 I/Os)

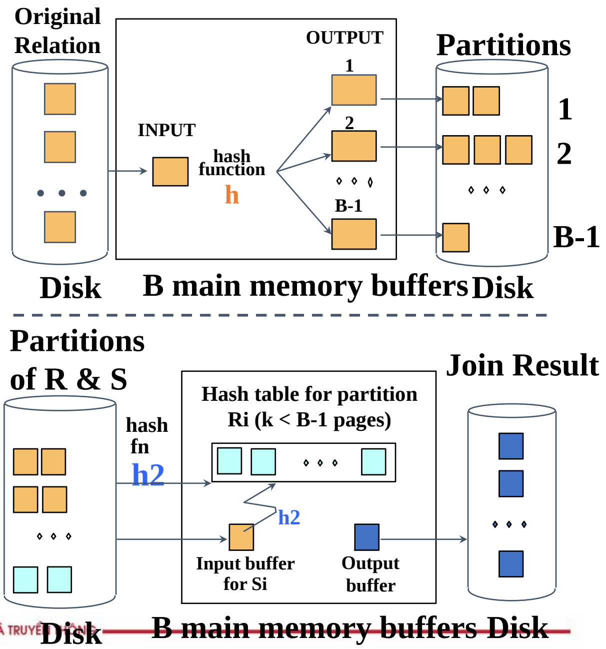
Refinement of Sort-Merge Join

- We can combine the merging phases in the *sorting* of R and S with the merging required for the join.
  - With B > $\sqrt{L}$, where *L* is the size of the larger relation, using the sorting refinement that produces runs of length 2B in Pass 0, #runs of each relation is < B/2.
  - Allocate 1 page per run of each relation, and `merge' while checking the join condition.
  - Cost: read+write each relation in Pass 0 + read each relation in (only) merging pass (+ writing of result tuples).
  - In example, cost goes down from 7500 to 4500 I/Os.
- In practice, cost of sort-merge join, like the cost of external sorting, is *linear*.

Hash-Join

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

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



Observations on Hash-Join

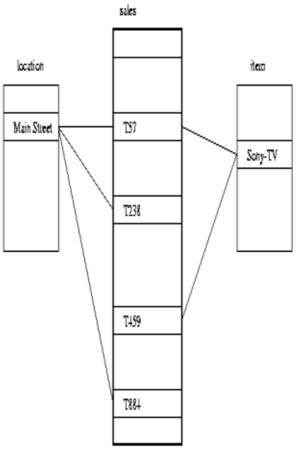
- #partitions k < B-1 (why?), and B-2 > size of largest partition to be held in memory. Assuming uniformly sized partitions, and maximizing k, we get:
  - k= B-1, and M/(B-1) < B-2, i.e., B must be > $\sqrt{M}$
- If we build an in-memory hash table to speed up the matching of tuples, a little more memory is needed.
- If the hash function does not partition uniformly, one or more R partitions may not fit in memory. Can apply hash-join technique recursively to do the join of this R-partition with corresponding S-partition.

Cost of Hash-Join

- In partitioning phase, read+write both relns; 2(M+N). In matching phase, read both relns; M+N I/Os.
- In our running example, this is a total of 4500 I/Os.
- Sort-Merge Join vs. Hash Join:
  - Given a minimum amount of memory (*what is this, for each?*) both have a cost of 3(M+N) I/Os. Hash Join superior on this count if relation sizes differ greatly. Also, Hash Join shown to be highly parallelizable.
  - Sort-Merge less sensitive to data skew; result is sorted.

Join Indices

- Traditional indices map the values to a list of record ids
  - It materializes relational join in JI file and speeds up relational join — a rather costly operation
- In data warehouses, join index relates the values of the dimensions of a start schema to rows in the fact table.
  - E.g. fact table: *Sales* and two dimensions *city* and *product*
    - A join index on *city* maintains for each distinct city a list of R-IDs of the tuples recording the Sales in the city
  - Join indices can span multiple dimensions



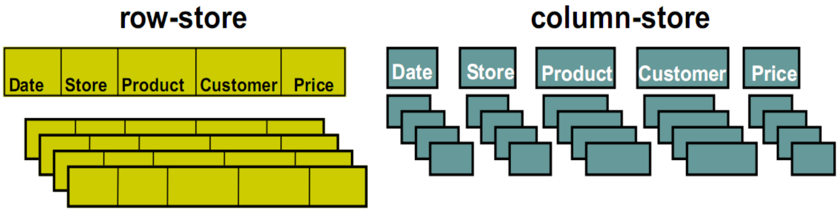
General Join Conditions

- Equalities over several attributes (e.g., *R.sid=S.sid* AND *R.rname=S.sname*):
  - For Index NL, build index on *<sid, sname>* (if S is inner); or use existing indexes on *sid* or *sname*.
  - For Sort-Merge and Hash Join, sort/partition on combination of the two join columns.
- Inequality conditions (e.g., *R.rname < S.sname*):
  - For Index NL, need (clustered!) B+ tree index.
    - Range probes on inner; # matches likely to be much higher than for equality joins.
  - Hash Join, Sort Merge Join not applicable.
  - Block NL quite likely to be the best join method here.

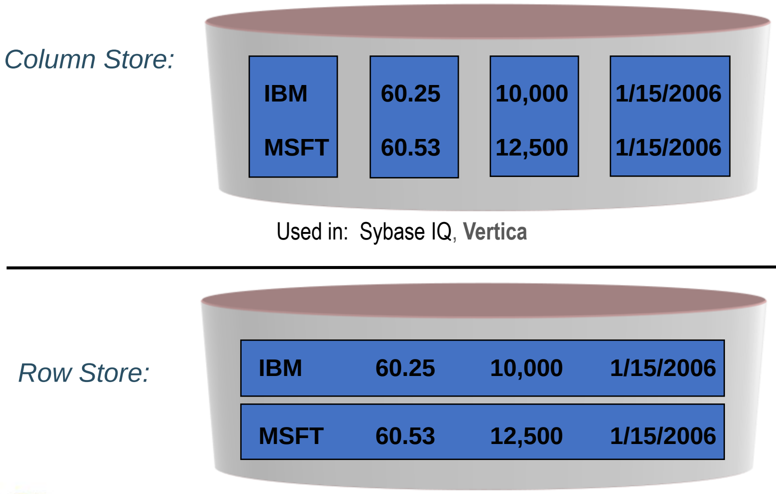
An invention in 2000s: Column Stores for OLAP

Row Store and Column Store

- In row store data are stored in the disk tuple by tuple.
- Where in column store data are stored in the disk column by column



Row Store vs Column Store



IBM MSFT

60.25 60.53

10,000 12,500

1/15/2006 1/15/2006

Used in: Sybase IQ, **Vertica**

IBM 60.25 10,000 1/15/2006

MSFT 60.53 12,500 1/15/2006

Used in: Oracle, SQL Server, DB2, Netezza,…

Row Store and Column Store

For example the query

```
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
```

Row-store: one row = 212 columns!

Column-store: 7 attributes

Row Store and Column Store

Row Store	Column Store
(+) Easy to add/modify a record	(+) Only need to read in relevant data
(-) Might read in unnecessary data	(-) Tuple writes require multiple accesses

- So column stores are suitable for read-mostly, read-intensive, large data repositories

Column Stores: High Level

- Read only what you need
  - “Fat” fact tables are typical
  - Analytics read only a few columns
- Better compression
- Execute on compressed data
- Materialized views help row stores and column stores about equally

Data model (Vertica/C-Store)

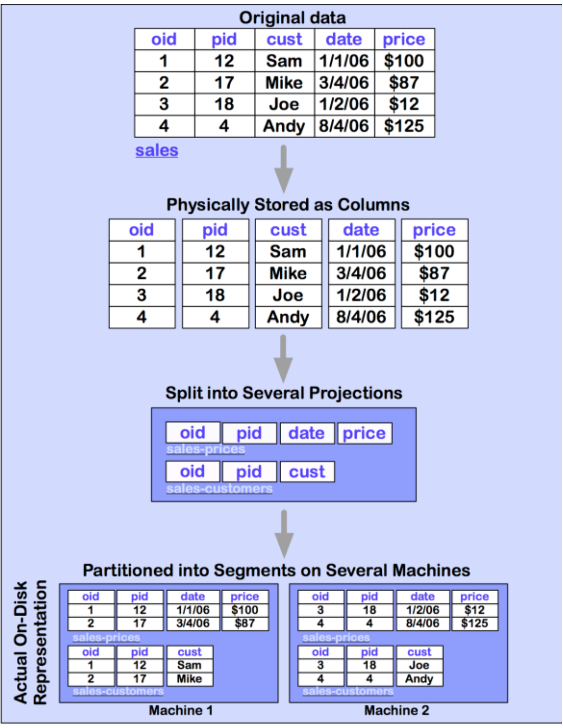
- Same as relational data model
  - Tables, rows, columns
  - Primary keys and foreign keys
  - **Projections**
    - From single table
    - Multiple joined tables
- Example

Normal relational model

EMP(name, age, dept, salary)
DEPT(dname, floor)

Possible C-store model

EMP1 (name, age)
EMP2 (dept, age, DEPT.floor)
EMP3 (name, salary)
DEPT1(dname, floor)



Original data

oid	pid	cust	date	price
1	12	Sam	1/1/06	$100
2	17	Mike	3/4/06	$87
3	18	Joe	1/2/06	$12
4	4	Andy	8/4/06	$125

sales

Physically Stored as Columns

oid	pid	cust	date	price
1	12	Sam	1/1/06	$100
2	17	Mike	3/4/06	$87
3	18	Joe	1/2/06	$12
4	4	Andy	8/4/06	$125

Split into Several Projections

Partitioned into Segments on Several Machines

Actual On-Disk Representation

Machine 1

oid	pid	date	price
1	12	1/1/06	$100
2	17	3/4/06	$87

sales-prices

oid	pid	cust
1	12	Sam
2	17	Mike

sales-customers

Machine 2

oid	pid	date	price
3	18	1/2/06	$12
4	4	8/4/06	$125

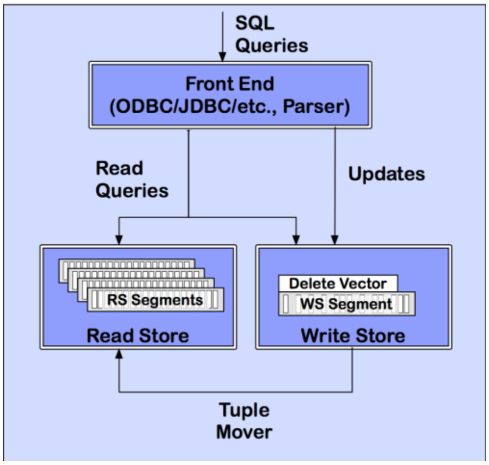
sales-prices

oid	pid	cust
3	18	Joe
4	4	Andy

sales-customers

C-Store/Vertica Architecture

(from vertica Technical Overview White Paper)



Read store: Column Encoding/Compression

- Use compression schemes and indices
  - Null Suppression
  - Dictionary encoding
  - Run Length encoding
  - Bit-Vector encoding

  - Self-order (key), few distinct values
    - (value, position, # items)
    - Indexed by clustered B-tree
  - Foreign-order (non-key), few distinct values
    - (value, bitmap index)
    - B-tree index: position → values
  - Self-order, many distinct values
    - Delta from the previous value
    - B-tree index
  - Foreign-order, many distinct values
    - Unencoded

Compression

- Trades I/O for CPU
  - Increased column-store opportunities:
  - Higher data value locality in column stores
  - Techniques such as run length encoding far more useful

Write Store

- Same structure, but explicitly use (segment, key) to identify records
  - Easier to maintain the mapping
  - Only concerns the inserted records
- Tuple mover
  - Copies batch of records to RS
- Delete record
  - Mark it on RS
  - Purged by tuple mover

How to solve read/write conflict

- Situation: one transaction updates the record X, while another transaction reads X.
- Use snapshot isolation

Query Execution - Operators

- **Select:** Same as relational algebra, but produces a bit string
- **Project:** Same as relational algebra
- **Join:** Joins projections according to predicates

- **Aggregation:** SQL like aggregates
- **Sort:** Sort all columns of a projection

Query Execution - Operators

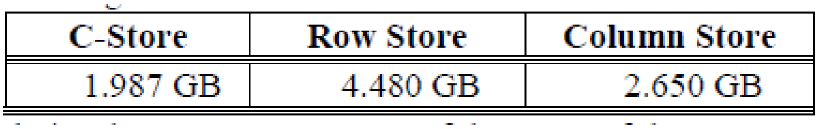
- **Decompress:** Converts compressed column to uncompressed representation
- **Mask**(Bitstring B, Projection Cs) => emit only those values whose corresponding bits are 1
- **Concat:** Combines one or more projections sorted in the same order into a single projection
- **Permute:** Permutes a projection according to the ordering defined by a join index
- **Bitstring operators:** Band – Bitwise AND, Bor – Bitwise OR, Bnot – complement

Benefits in query processing

- Selection – has more indices to use
- Projection – some "projections" already defined
- Join – some projections are materialized joins
- Aggregations – works on required columns only

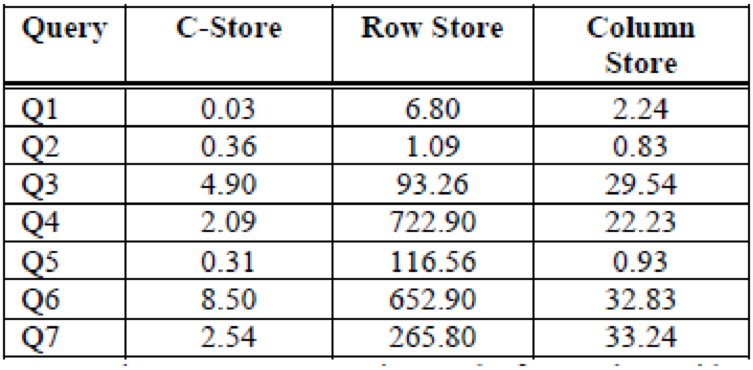
Evaluation

- Use TPC-H – decision support queries
- Storage



C-Store	Row Store	Column Store
1.987 GB	4.480 GB	2.650 GB

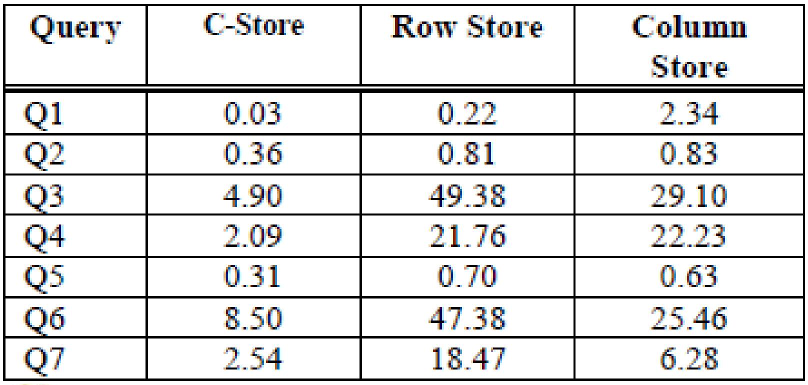
Query performance



Query	C-Store	Row Store	Column Store
Q1	0.03	6.80	2.24
Q2	0.36	1.09	0.83
Q3	4.90	93.26	29.54
Q4	2.09	722.90	22.23
Q5	0.31	116.56	0.93
Q6	8.50	652.90	32.83
Q7	2.54	265.80	33.24

Query performance

- Row store uses materialized views



Query	C-Store	Row Store	Column Store
Q1	0.03	0.22	2.34
Q2	0.36	0.81	0.83
Q3	4.90	49.38	29.10
Q4	2.09	21.76	22.23
Q5	0.31	0.70	0.63
Q6	8.50	47.38	25.46
Q7	2.54	18.47	6.28

Summary: the performance gain

- Column representation – avoids reads of unused attributes
- Storing overlapping projections – multiple orderings of a column, more choices for query optimization
- Compression of data – more orderings of a column in the same amount of space
- Query operators operate on compressed representation

VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

104

ĐẠI HỌC BÁCH KHOA HÀ NỘI
VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

Google's Dremel/Big Query Interactive Analysis of Web-Scale Datasets

105

105

Big Query system

- Trillion-record, multi-terabyte datasets at interactive speed
  - Scales to thousands of nodes
  - Fault and straggler tolerant execution
- Nested data model
  - Complex datasets; normalization is prohibitive
  - Columnar storage and processing
- Tree architecture (as in web search)
- Interoperates with Google's data mgmt tools
  - *In situ* data access (e.g., GFS, Bigtable)
  - MapReduce pipelines

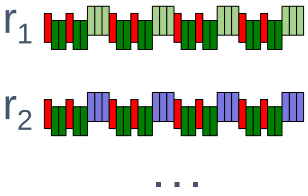
Widely used inside Google

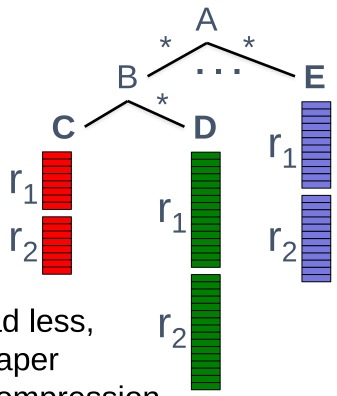
- Analysis of crawled web documents
- Tracking install data for applications on Android Market
- Crash reporting for Google products
- OCR results from Google Books
- Spam analysis
- Debugging of map tiles on Google Maps

- Tablet migrations in managed Bigtable instances
- Results of tests run on Google's distributed build system
- Disk I/O statistics for hundreds of thousands of disks
- Resource monitoring for jobs run in Google's data centers
- Symbols and dependencies in Google's codebase

Records vs. columns

```
DocId: 10                    r1
Links
  Forward: 20
Name
  Language
    Code: 'en-us'
    Country: 'us'
  Url: 'http://A'
Name
  Url: 'http://B'
```





Read less,
cheaper
decompression

Challenge: preserve structure, reconstruct from a subset of fields

Nested data model

http://code.google.com/apis/protocolbuffers

```
                                              multiplicity:
message Document {
  required int64 DocId;                         [1,1]
  optional group Links {
    repeated int64 Backward;                    [0,*]
    repeated int64 Forward;
  }
  repeated group Name {
    repeated group Language {
      required string Code;
      optional string Country;                  [0,1]
    }
    optional string Url;
  }
}
```

```
DocId: 10                    r1
Links
  Forward: 20
  Forward: 40
  Forward: 60
Name
  Language
    Code: 'en-us'
    Country: 'us'
  Language
    Code: 'en'
  Url: 'http://A'
Name
  Url: 'http://B'
Name
  Language
    Code: 'en-gb'
    Country: 'gb'
```

```
DocId: 20                    r2
Links
  Backward: 10
  Backward: 30
  Forward:  80
Name
  Url: 'http://C'
```

Column-striped representation

DocId

value	r	d
10	0	0
20	0	0

Name.Url

value	r	d
http://A	0	2
http://B	1	2
NULL	1	1
http://C	0	2

Links.Forward

Links.Backward

value	r	d
NULL 20	0	1 2
10 40	0 1	2
30 60	1	2
80	0	2

Name.Language.Code

value	r	d
en-us	0	2
en	2	2
NULL	1	1
en-gb	1	2
NULL	0	1

Name.Language.Country

value	r	d
us	0	3
NULL	2	2
NULL	1	1
gb	1	3
NULL	0	1

Repetition and definition levels

r=1 r=2 (non-repeating)

Name.Language.Code

value	r	d
en-us	0	2
en	2	2
NULL	1	1
en-gb	1	2
NULL	0	1

record (r=0) has repeated

Language (r=2) has repeated

r_1

```
DocId: 10
Links
  Forward: 20
  Forward: 40
  Forward: 60
Name
  Language
    Code: 'en-us'
    Country: 'us'
  Language
    Code: 'en'
  Url: 'http://A'
Name
  Url: 'http://B'
Name
  Language
    Code: 'en-gb'
    Country: 'gb'
```

r_2

```
DocId: 20
Links
  Backward: 10
  Backward: 30
  Forward:  80
Name
  Url: 'http://C'
```

r: At what repeated field in the field's path the value has repeated

d: How many fields in paths that could be undefined (opt. or rep.) are actually present

Query processing

- Optimized for select-project-aggregate
  - Very common class of interactive queries
  - Single scan
  - Within-record and cross-record aggregation
- Approximations: count(distinct), top-k
- Joins, temp tables, UDFs/TVFs, etc.

SQL dialect for nested data

```
SELECT DocId AS Id,
  COUNT(Name.Language.Code) WITHIN Name AS Cnt,
  Name.Url + ',' + Name.Language.Code AS Str
FROM t
WHERE REGEXP(Name.Url, '^http') AND DocId < 20;
```

Output table

t_1

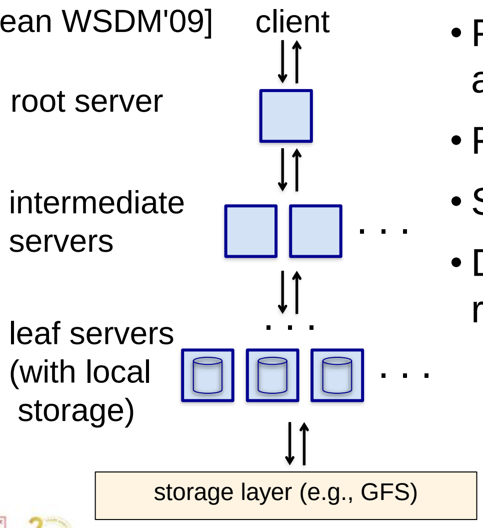
```
Id: 10
Name
  Cnt: 2
  Language
    Str: 'http://A,en-us'
    Str: 'http://A,en'
Name
  Cnt: 0
```

Output schema

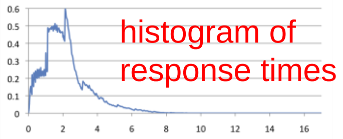
```
message QueryResult {
  required int64 Id;
  repeated group Name {
    optional uint64 Cnt;
    repeated group Language {
      optional string Str;
    }
  }
}
```

Serving tree

[Dean WSDM'09]

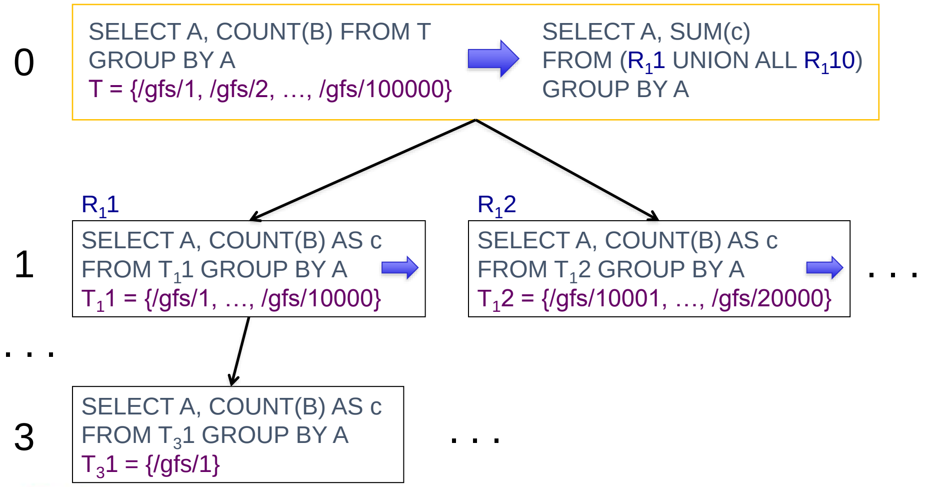


- Parallelizes scheduling and aggregation
- Fault tolerance
- Stragglers
- Designed for "small" results (<1M records)



histogram of response times

Example: count()



0

SELECT A, COUNT(B) FROM T
GROUP BY A
T = {/gfs/1, /gfs/2, …, /gfs/100000}

SELECT A, SUM(c)
FROM ($R_1$1 UNION ALL $R_1$10)
GROUP BY A

1

$R_1$1

SELECT A, COUNT(B) AS c
FROM $T_1$1 GROUP BY A
$T_1$1 = {/gfs/1, …, /gfs/10000}

$R_1$2

SELECT A, COUNT(B) AS c
FROM $T_1$2 GROUP BY A
$T_1$2 = {/gfs/10001, …, /gfs/20000}

. . .

. . .

3

SELECT A, COUNT(B) AS c
FROM $T_3$1 GROUP BY A
$T_3$1 = {/gfs/1}

. . .

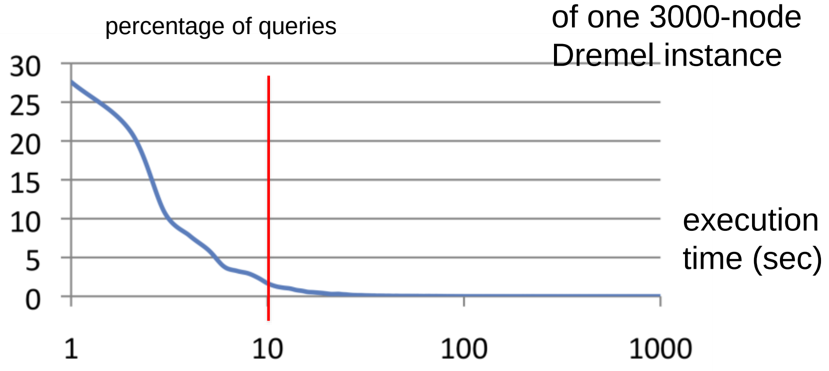
Data access ops

Experiments

- 1 PB of real data
  (uncompressed, non-replicated)
- 100K-800K tablets per table
- Experiments run during business hours

Table name	Number of records	Size (unrepl., compressed)	Number of fields	Data center	Repl. factor
T1	85 billion	87 TB	270	A	3 ×
T2	24 billion	13 TB	530	A	3 ×
T3	4 billion	70 TB	1200	A	3 ×
T4	1+ trillion	105 TB	50	B	3 ×
T5	1+ trillion	20 TB	30	B	2 ×

Interactive speed



Monthly query workload
of one 3000-node Dremel instance

Most queries complete under 10 sec

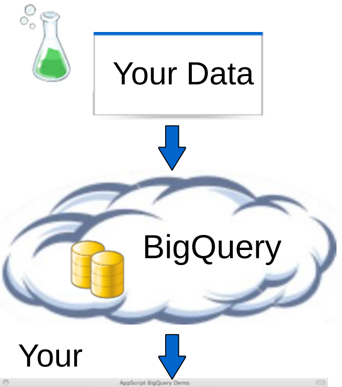
BigQuery: powered by Dremel

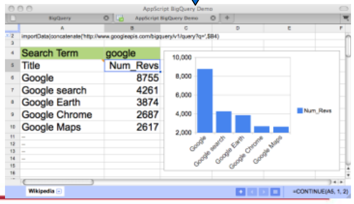
http://code.google.com/apis/bigquery/

1. Upload — Upload your data to Google Storage

2. Process — Import to tables

3. Act — Run queries





importData(concatenate('http://www.googleapis.com/bigquery/v1/query?q=',$B4))

Search Term	google
Title	Num_Revs
Google	8755
Google search	4261
Google Earth	3874
Google Chrome	2687
Google Maps	2617

=CONTINUE(A5, 1, 2)

List of Column Databases

- Vertica/C-Store
- SybaseIQ
- MonetDB
- LucidDB
- HANA
- Google’s Dremel
- Parcell-> Redshit (Another Cloud-DB Service)

Take-home messages

- OLAP
  - Multi-relational Data model
  - Operators
  - SQL
- Data warehouse (architecture, issues, optimizations)
- Join Processing
- Column Stores (Optimized for OLAP workload)

ĐẠI HỌC
BÁCH KHOA

25 YEARS ANNIVERSARY
SOICT

HA NOI UNIVERSITY OF SCIENCE AND TECHNOLOGY
SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY

VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG
SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY

Thank you
for your
attentions!

soict.hust.edu.vn/ fb.com/groups/soict

122

ĐẠI HỌC BÁCH KHOA HÀ NỘI
VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

123