CSE 544
Principles of Database Management Systems

Fall 2016

Lecture 14 - Data Warehousing and Column Stores

References

- Data Cube: A Relational Aggregation Operator Generalizing Group By, Cross-Tab, and Sub-Totals.

  Jim Gray et. al. Data Mining and Knowledge Discovery 1, 29-53. 1997

- Database management systems.

  Ramakrishnan and Gehrke.

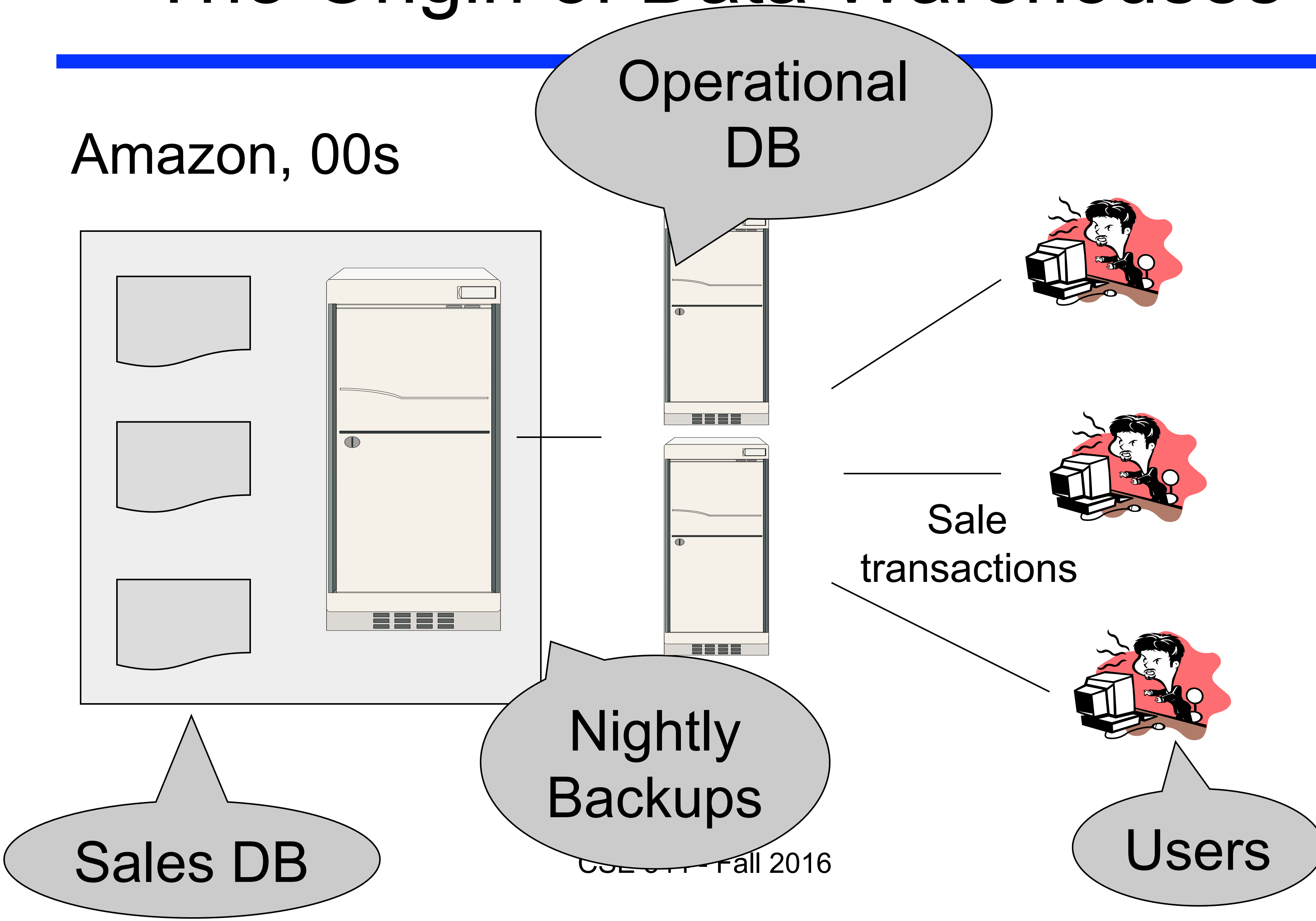
  Third Ed. **Chapter 25**

Why Data Warehouses?

- Production DBMSs designed to manage operational data
  - Goal: support every day activities
  - Online transaction processing (OLTP)
  - Ex: Tracking sales and inventory of each Wal-mart store
- Data Warehouse designed to analyze and explore data
  - Goal: summarize and discover trends to support decision making
  - Online analytical processing (OLAP)
- Data warehouse usually updated overnight from production databases

The Origin of Data Warehouses

Amazon, 00s



Data Warehouse Overview

- Consolidated data from many sources
  - Must create a single unified schema
  - The warehouse is like a materialized view
- Very large size: terabytes of data are common
- Complex read-only queries (no updates)
- Fast response time is (not as) important
  - Compared to transaction processing

Star Schema

- Central table, e.g.
  - SALES(<u>saleID</u>, time, price, storeID, productID, ...)

- Dimension tables, e.g.
  - Store(<u>storeID</u>, sname, location, ...),
  - Product(<u>productID</u>, pname, weight, ...)
  - SalesPerson(<u>personID</u>, name, ...)
  - ...

OLAP queries

- ETL pipeline load data into a data warehouse
- Operators:
  - Rollup
  - Drill down
  - Pivoting
  - Cube

The ETL Pipeline

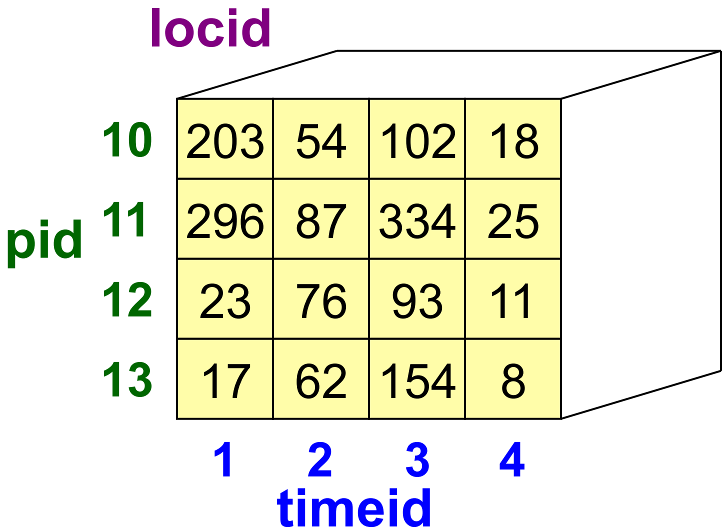
- **Extract** data from distributed operational databases
- **Clean** to minimize errors and fill in missing information
- **Transform** to reconcile semantic mismatches
  - Performed by defining views over the data sources
- **Load** to materialize the above defined views
  - Build indexes and additional materialized views
- **Refresh** to propagate updates to warehouse periodically

Back to Warehouses: Outline

- Multidimensional data model and operations
- Data cube & rollup operators
- Data warehouse implementation issues
- Other extensions for data analysis

Multidimensional Data Model

- Focus of the analysis is a collection of measures
  - Example: Wal-mart sales
- Each measure depends on a set of dimensions
  - Example: product (pid), location (lid), and time of the sale (timeid)



Slicing: equality selection on one or more dimensions

Dicing: range selection

Star Schema

Representing multidimensional data as relations (ROLAP)

Product

pid	pname	category	price

Location

locid	city	state	country

Facts table: Sales

In BCNF

Sales

sid	pid	timeid	locid	amount

Dimensions tables

- Product
- Location
- Times

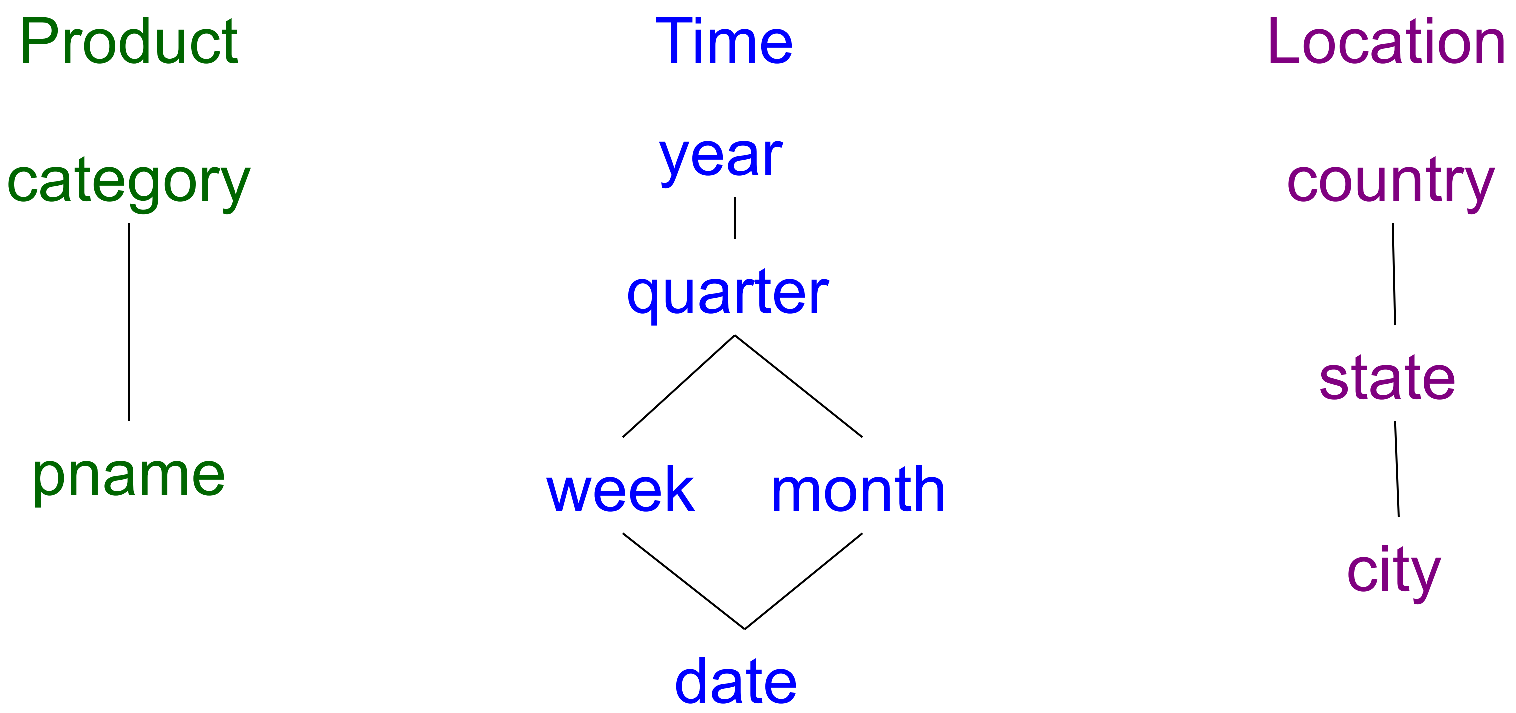
Not necessarily normalized

timeid	date	week	month	quarter	year

Times

Dimension Hierarchies

Dimension values can form a hierarchy described by attributes



Desired Operations

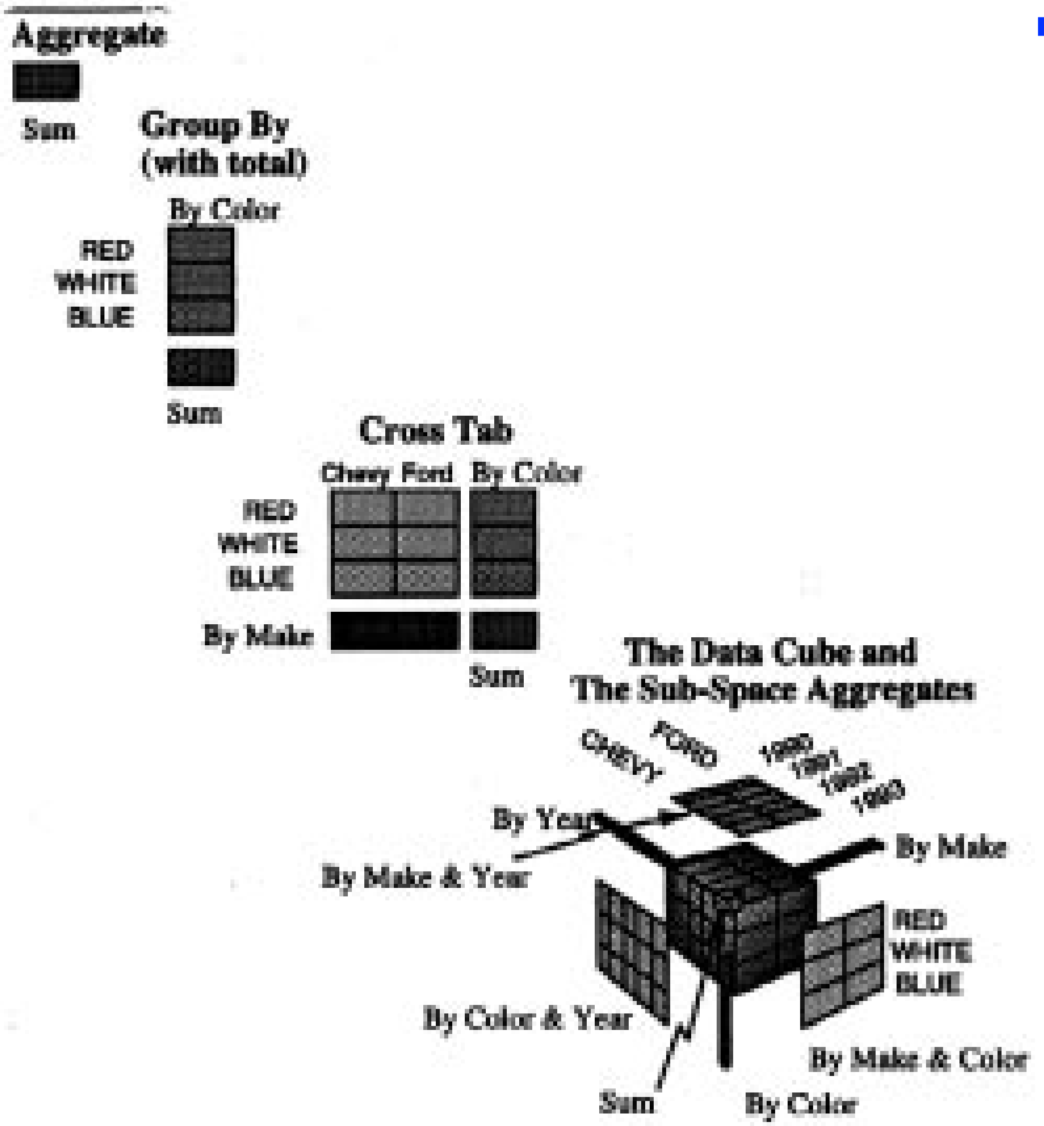
- Histograms (agg. over computed categories) (paper p.34)
- Summarize at different levels: roll-up and drill-down
  - Ex: total sales by day, week, quarter, and year
- Pivoting
  - Ex: pivot on location and time
  - Result of pivot is a cross-tabulation
  - Column values become labels

	WI	CA	Total
2005	500	200	**700**
2006	150	850	**1000**
2007	250	400	**650**
Total	**900**	**1450**	**2350**

Challenge 1: Representation

- Problem: How to represent multi-level aggregation?
  - Ex: Table 3 in the paper need 2^N columns for N dimensions!
  - Ex: Table 4 has even more columns!

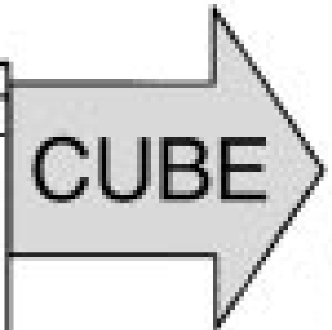
Challenge 1: Representation



```
SELECT Model, Year, Color, SUM(sales) AS Sales
FROM Sales
WHERE Model in ('Ford', 'Chevy')
  AND Year BETWEEN 1990 AND 1992
GROUP BY CUBE Model, Year, Color;
```

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
Chevy	1990	blue	62
Chevy	1990	red	5
Chevy	1990	white	87
Chevy	1990	ALL	154
Chevy	1991	blue	49
Chevy	1991	red	54
Chevy	1991	white	95
Chevy	1991	ALL	198
Chevy	1992	blue	71
Chevy	1992	red	31
Chevy	1992	white	54
Chevy	1992	ALL	156
Chevy	ALL	blue	182
Chevy	ALL	red	90
Chevy	ALL	white	236
Chevy	ALL	ALL	508
Ford	1990	blue	63
Ford	1990	red	64
Ford	1990	white	62
Ford	1990	ALL	189
Ford	1991	blue	55
Ford	1991	red	52
Ford	1991	white	9
Ford	1991	ALL	116
Ford	1992	blue	39
Ford	1992	red	27
Ford	1992	white	62
Ford	1992	ALL	128
Ford	ALL	blue	157
Ford	ALL	red	143
Ford	ALL	white	133
Ford	ALL	ALL	433
ALL	1990	blue	125
ALL	1990	red	69
ALL	1990	white	149
ALL	1990	ALL	343
ALL	1991	blue	106
ALL	1991	red	104
ALL	1991	white	110
ALL	1991	ALL	314
ALL	1992	blue	110
ALL	1992	red	58
ALL	1992	white	116
ALL	1992	ALL	284
ALL	ALL	blue	339
ALL	ALL	red	233
ALL	ALL	white	369
ALL	ALL	ALL	941

Challenge 1: Representation

- Problem: How to represent multi-level aggregation?
  - Ex: Table 3 in the paper need 2^N columns for N dimensions!
  - Ex: Table 4 has even more columns!
  - And that's without considering any hierarchy on the dimensions!
- Solution: special "all" value

T.year	L.state	SUM(S.sales)
2005	WI	**500**
2005	CA	**200**
2005	ALL	**700**
...	...	**...**
ALL	ALL	**2350**

Note: SQL-1999 standard uses NULL values instead of ALL

Challenge 2: Computing Aggregations

- Need 2^N different SQL queries to compute all aggregates
  - Expressing roll-up of a single column and cross-table queries is thus daunting
  - Cannot optimize all these independent queries

- Solution: CUBE and ROLLUP operators

Outline

- Multidimensional data model and operations
- Data cube & rollup operators
- Data warehouse implementation issues
- Other extensions for data analysis

Data Cube

- CUBE is the N-dimensional generalization of aggregate

- Cube in SQL-1999

```
SELECT T.year, L.state, SUM(S.sales)
FROM Sales S, Times T, Locations L
WHERE S.timeid=T.timeid and S.locid=L.locid
GROUP BY CUBE (T.year,L.state)
```

- Creating a data cube requires generating the power set of the aggregation columns

Rollup

- Rollup produces a subset of a cube

- Rollup in SQL-1999

```
SELECT T.year,T.quarter, SUM(S.sales)
FROM Sales S, Times T
WHERE S.timeid=T.timeid
GROUP BY ROLLUP (T.year,T.quarter)
```

- Will aggregate over each pair of (year,quarter), each year, and total, but will **not** aggregate over each quarter

Computing Cubes and Rollups

- Naive algorithm
  - For each new tuple, update each of 2^N matching cells
- More efficient algorithm
  - Use intermediate aggregates to compute others
  - Relatively easy for distributive and algebraic functions
- Updating a cube in response to updates is more challenging

Outline

- Multidimensional data model and operations
- Data cube & rollup operators
- Data warehouse implementation issues
- Other extensions for data analysis

Indexes

- Bitmap indexes: good for sparse attributes (few values)

M	F
0	1
1	0
1	0

custid	name	gender	rating
10	Alice	F	3
11	Bob	M	4
12	Chuck	M	1

1	2	3	4
0	0	1	0
0	0	0	1
1	0	0	0

- Join indexes: to speed-up specific join queries
  - Example: Join fact table F with dimension tables D1 and D2
  - Index contain triples of rids $<r_1,r_2,r>$ from D_1, D_2, and F that join
  - Alternatively, two indexes, each one with pairs $<v_1,r>$ or $<v_2,r>$ where v_1, v_2 are values of tuples from D_1, D_2 that join with r

Materialized Views

- How to choose views to materialize?
  – Physical database tuning

- How to keep view up-to-date?
  – Could recompute entire view for each update: expensive
  – Better approach: incremental view maintenance
  – Example: recompute only affected partition

  – How often to synchronize? Periodic updates (at night) are typical
    - Think back in the case of Walmart

Outline

- Multidimensional data model and operations
- Data cube & rollup operators
- Data warehouse implementation issues
- Other extensions for data analysis

Additional Extensions for Decision Support

- Window queries

```
SELECT L.state, T.month, AVG(S.sales) over W AS movavg
FROM Sales S, Times T, Locations L
WHERE S.timeid = T.timeid AND S.locid=L.locid
WINDOW W AS (PARTITION BY L.State
             ORDER BY T.month
             RANGE BETWEEN INTERVAL '1' MONTH PRECEDING
             AND INTERVAL '1' MONTH FOLLOWING)
```

- Top-k queries: optimize queries to return top k results

- Online aggregation: produce results incrementally

Leveraging Column Stores

References

- The Design and Implementation of Modern Column-Oriented Database Systems Daniel Abadi, Peter Boncz, Stavros Harizopoulos, Stratos Idreos, Samuel Madden. Foundations and Trends® in Databases (Vol 5, Issue 3, 2012, pp 197-280).

Column-Oriented Databases

- Main idea:
  - Physical storage: complete vertical partition; each column stored separately: R.A, R.B, R.A
  - Logical schema: remains the same R(A,B,C)

- Main advantage:
  - Improved transfer rate: disk to memory, memory to CPU, better cache locality
  - Other advantages (next)

Data Layout

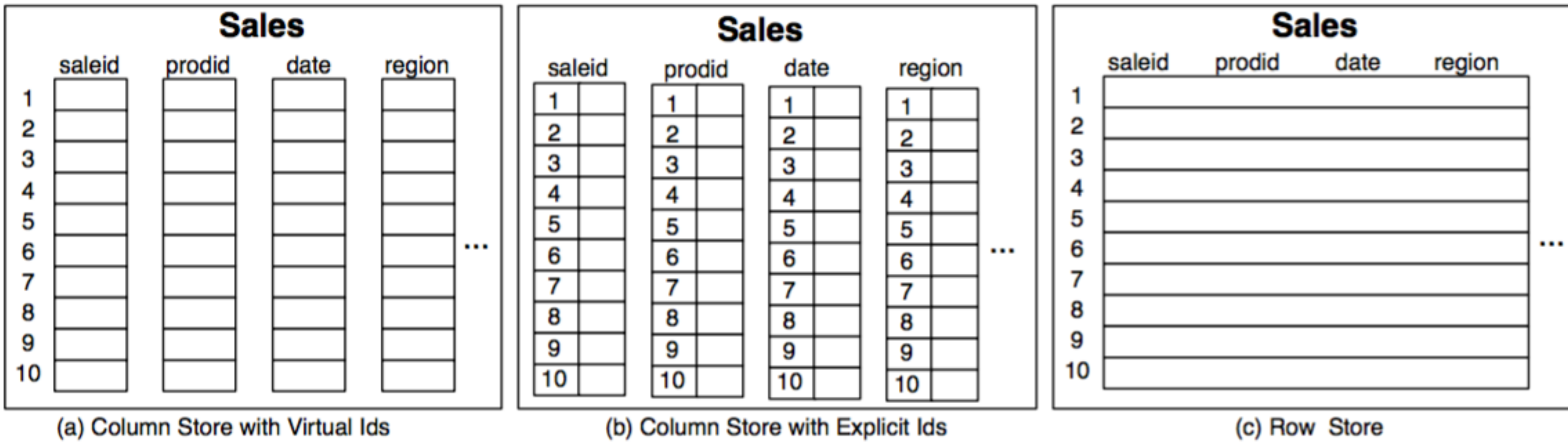


Figure 1.1: Physical layout of column-oriented vs row-oriented databases.

Basic tradeoffs:

- Reading all attributes of one records, v.s.
- Reading some attributes of many records

Key Architectural Trends (Sec.1)

- Virtual IDs
- Block-oriented and vertical processing
- Late materialization
- Column-specific compression

Key Architectural Trends (Sec.1)

- Virtual IDs
  - Offsets (arrays) instead of keys
- Block-oriented and vertical processing
  - Iterator model: one tuple→one block of tuples
- Late materialization
  - Postpone tuple reconstruction in query plan
- Column-specific compression
  - Much better than row-compression (why?)

Fig. 1.2

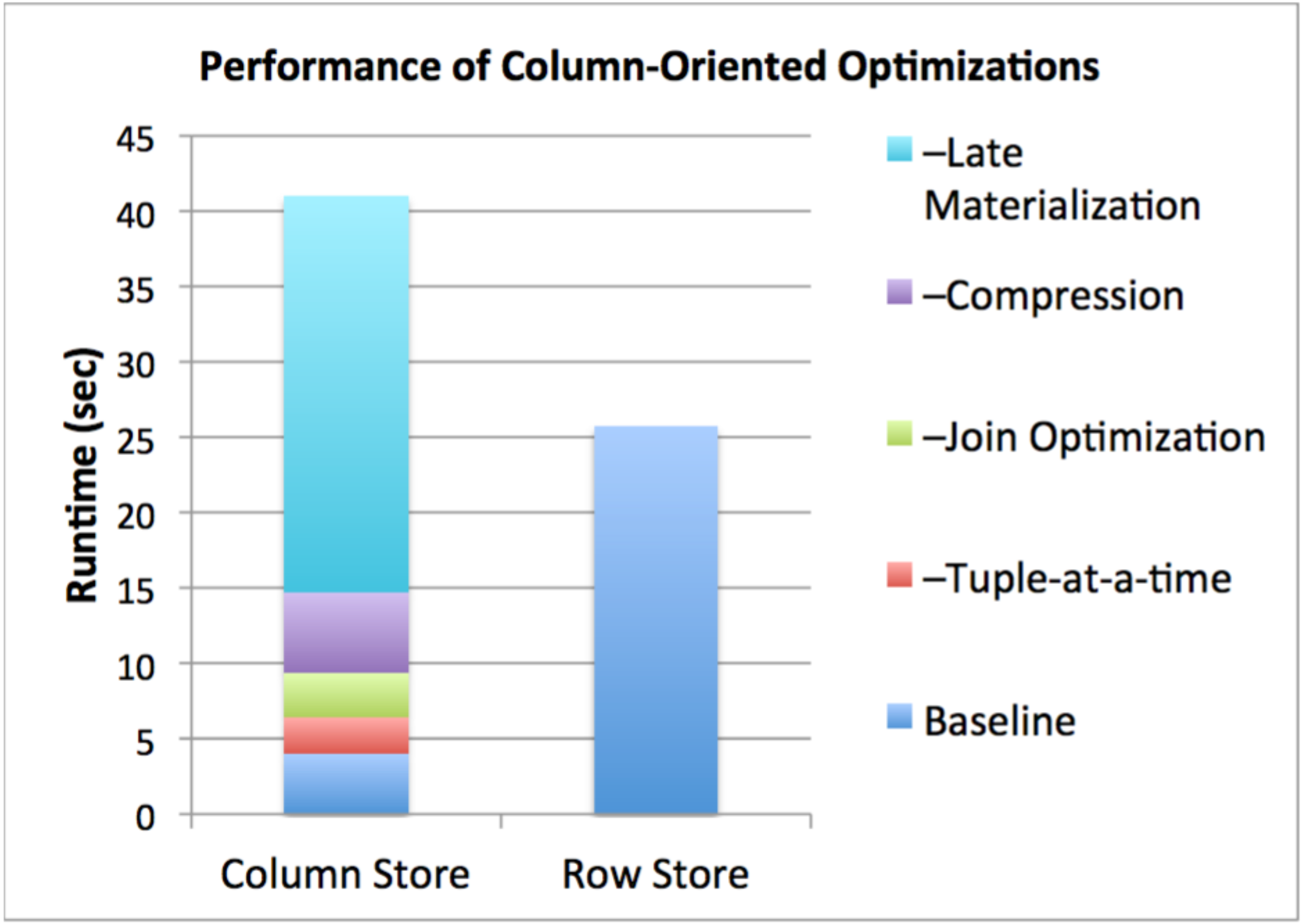


Figure 1.2: Performance of C-Store versus a commercial database system on the SSBM benchmark, with different column-oriented optimizations enabled.

Vectorized Processing

Review:

- Volcano-style iterator model
  - Next() method
  - Pipelining
- Materialization of all intermediate results
- Discuss in class:

```
select avg(A) from R where A < 100
```

Vectorized Processing

- Vectorized processing:
  - Next() returns a block of tuples (e.g. N=1000) instead of single tuple
- Pros:
  - No more large intermediate results
  - Tight inner loop for selection and/or avg
- Discuss in class:

select avg(A) from R where A < 100

Compression (Sec. 4)

- What is the advantage of compression in databases?
- Discuss main column-at-a-time compression techniques

Compression (Sec. 4)

- What is the advantage of compression in databases?

- Discuss main column-at-a-time compression techniques
  - Row-length encoding: F,F,F,F,M,M→4F,2M
  - Bit-vector (see also bit-map indexes)
  - Dictionary. More generally: Ziv-Lempel

Compression (Sec. 4)

Row-based
(4 pages)

Page

A	1
A	2

A	2
A	2

B	2
B	4

C	4
C	4

Column-based
(4 pages)

A
A
A
A

1
2
2
2

B
B
C
C

2
4
4
4

Page

Compressed
(2 pages)

4XA
2XB
2XC

1X1
4X2
5X4

Late Materialization (Sec. 4)

- What is it?
- Discuss $\Pi_B(\sigma_{A='a' \wedge D='d'}(R(A,B,C,D,\ldots))$

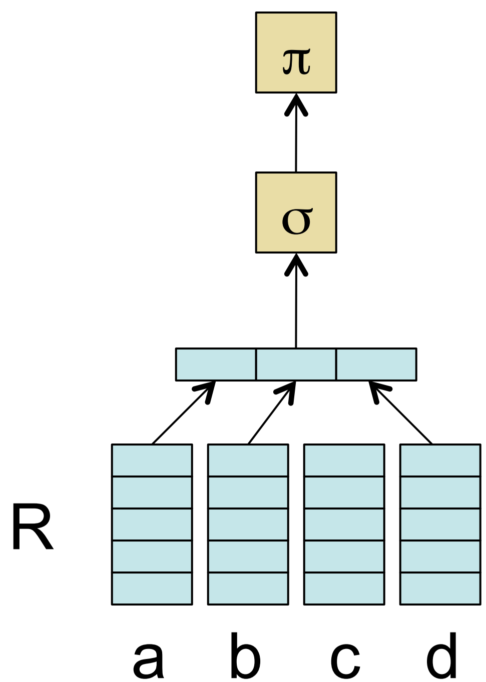
Late Materialization (Sec. 4)

- What is it?
- Discuss $\Pi_B(\sigma_{A=\text{'a'} \wedge D=\text{'d'}}(R(A,B,C,D,\ldots)))$
- Early materialization:
  - Retrieve positions with 'a' in column A: 2, 4, 5, 9, 25…
  - Retrieve those values in column D: 'x', 'd', 'y', 'd', 'd',...
  - Retain only positions with 'd': 4, 9, ...
  - Lookup values in column B: B[4], B[9], …
- Late materialization
  - Retrieve positions with 'a' in column A: 2, 4, 5, 9, 25…
  - Retrieve positions with 'd' in column D: 3, 4, 7, 9,12,..
  - Intersect: 4, 9, …
  - Lookup values in column B: B[4], B[9], …

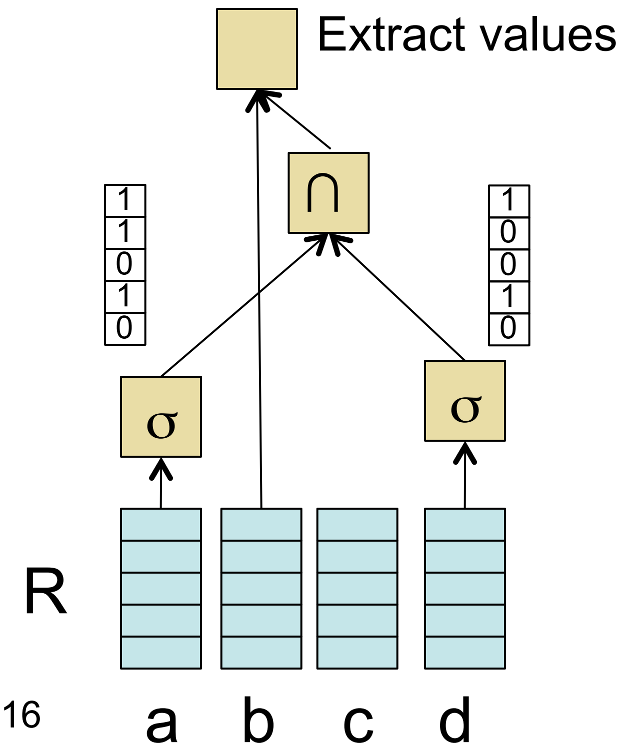
Late Materialization (Sec. 4)

Ex: `SELECT R.b from R where R.a=X and R.d=Y`

Early materialization



Late materialization



Joins (Sec. 4)

The result of a join R.A ⋈ S.A is an array of positions in R.A and S.A.
Note: sorted on R.A only.

R.A

1	Value42
2	Value36
3	Value42
4	Value44
5	Value38

⋈

S.A

1	Value38
2	Value42
3	Value46
4	Value36

=

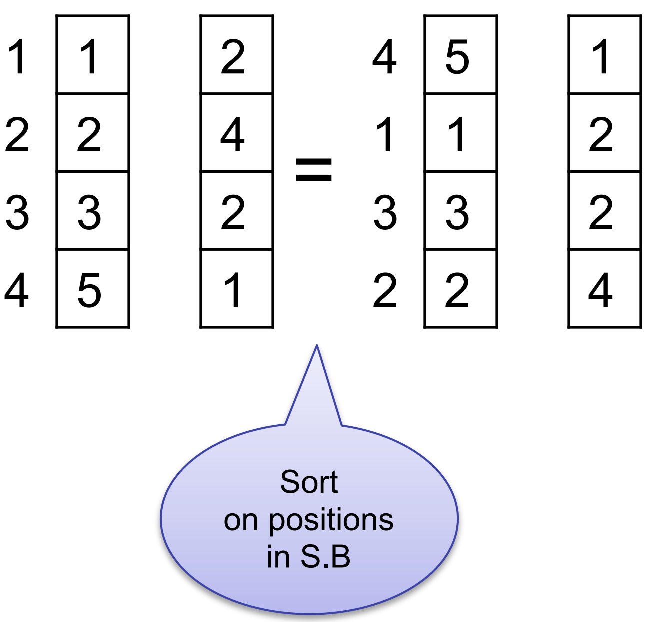
	Positions in R.A (sorted)	Positions in S.A (unsorted)
1	1	2
2	2	4
3	3	2
4	5	1

Jive-Join (Sec. 4)

Problem: accessing the values in the second table has poor memory locality

Solution: re-sort by the second column, fetch, sort back

E.g. $\Pi_{S.C}(R(A,\ldots) \bowtie S(B,C,\ldots))$

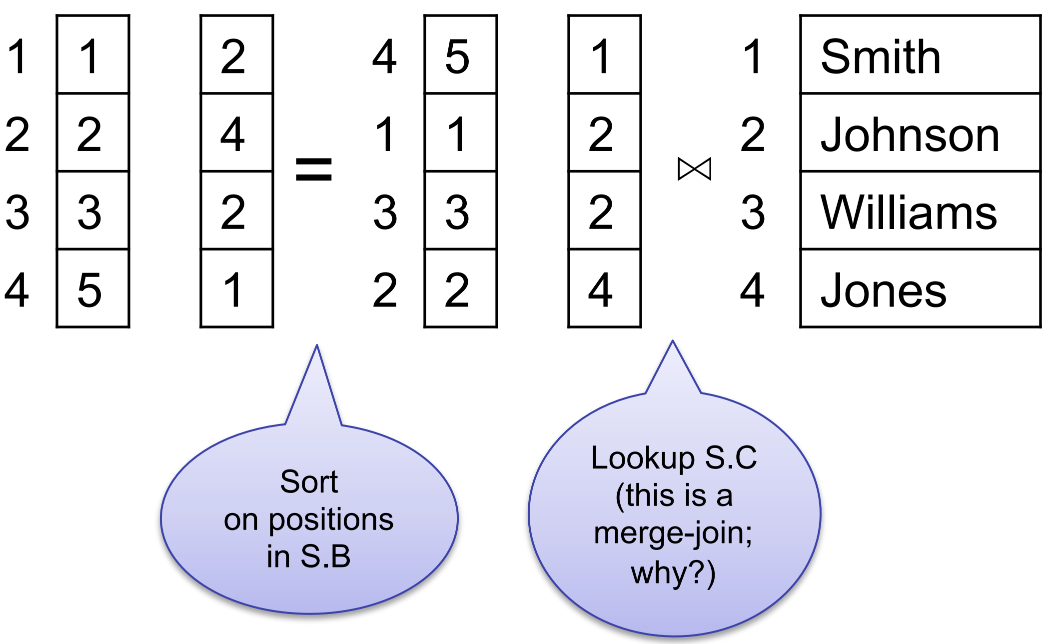


Jive-Join (Sec. 4)

Problem: accessing the values in the second table has poor memory locality

Solution: re-sort by the second coljun, fetch, sort back

E.g. $\Pi_{S.C}(R(A,\ldots) \bowtie S(B,C,\ldots\ ))$

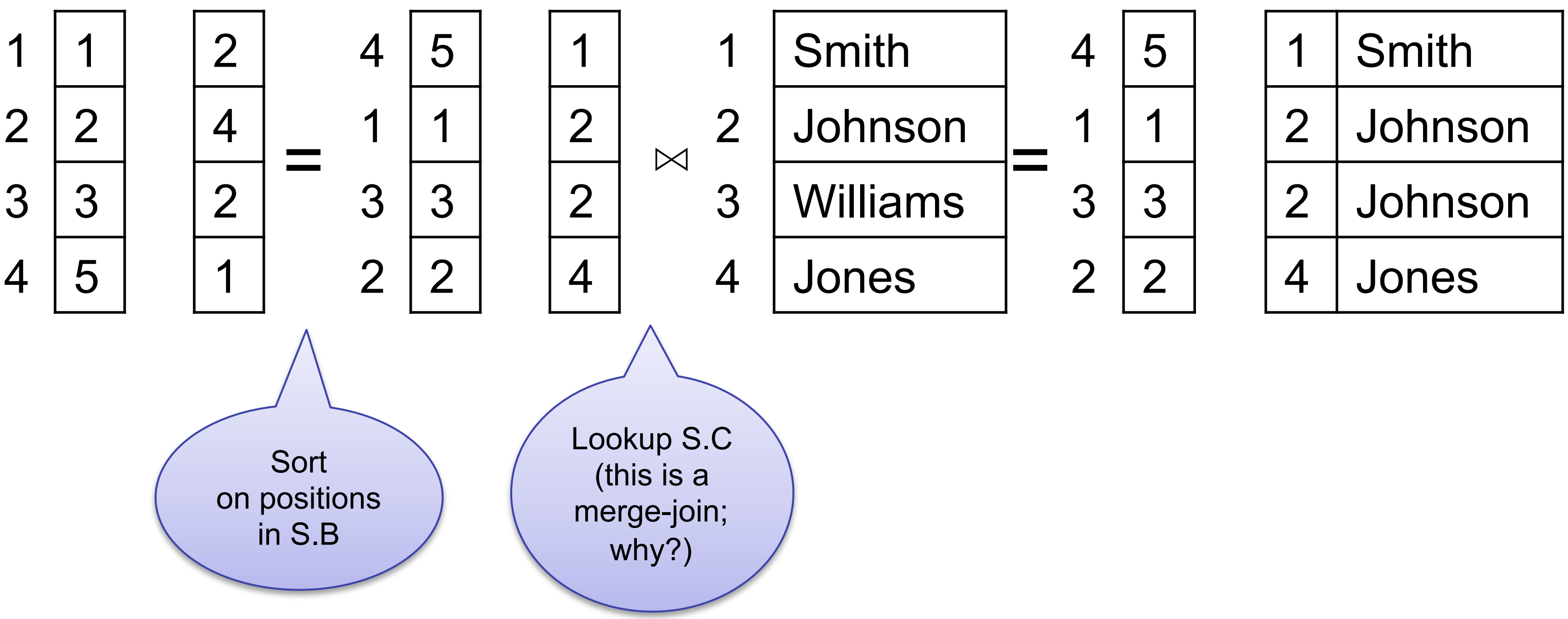


Jive-Join (Sec. 4)

Problem: accessing the values in the second table has poor memory locality

Solution: re-sort by the second coljun, fetch, sort back

E.g. $\Pi_{S.C}(R(A,\ldots) \bowtie S(B,C,\ldots\ ))$

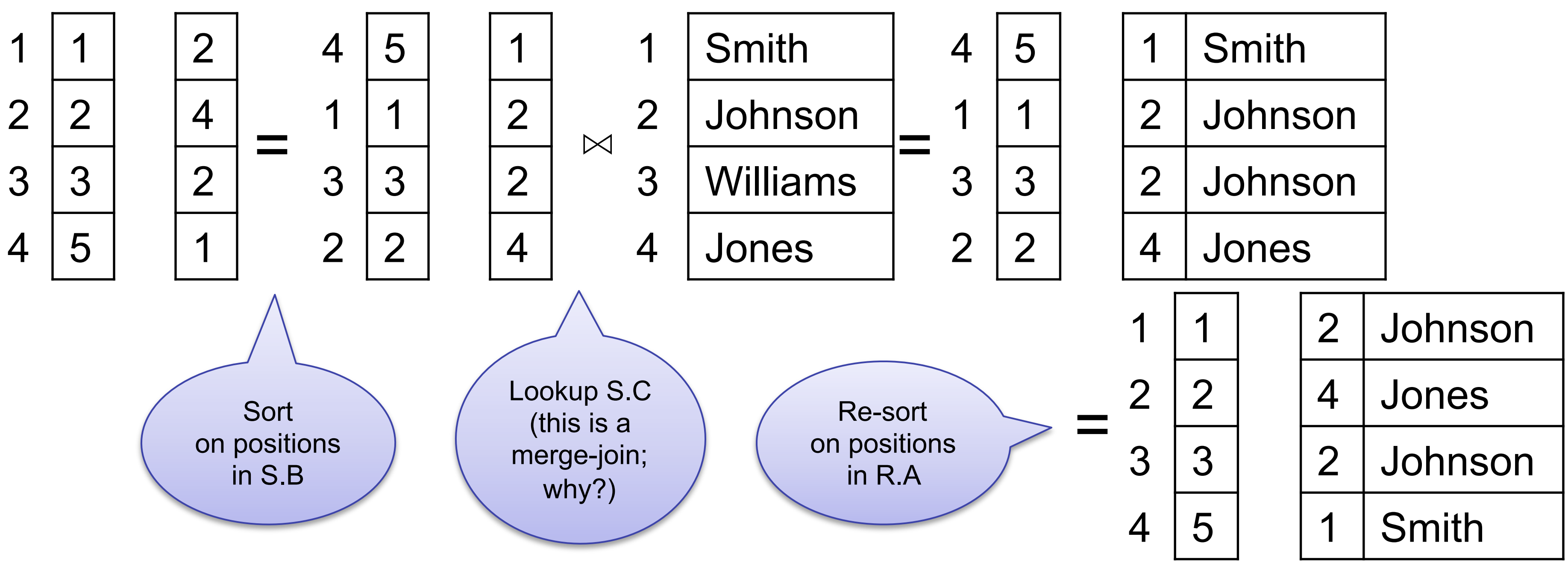


Jive-Join (Sec. 4)

Problem: accessing the values in the second table has poor memory locality

Solution: re-sort by the second coljun, fetch, sort back

E.g. $\Pi_{S.C}(R(A,\ldots) \bowtie S(B,C,\ldots\ ))$



Late Materialization

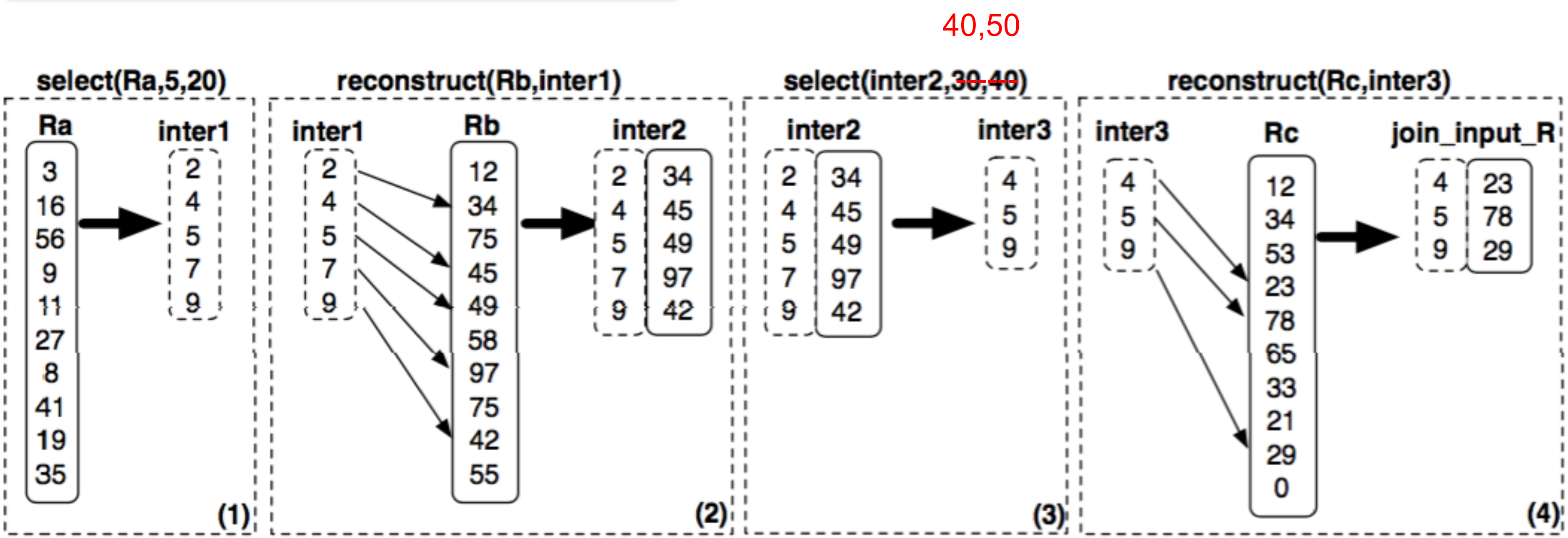
```
select sum(R.a) from R, S
where R.c = S.b
  and 5<R.a<20 and 40<R.b<50
 and 30<S.a<40
```

Initial Status

Relation R			Relation S	
Ra	Rb	Rc	Sa	Sb
3	12	12	17	11
16	34	34	49	35
56	75	53	58	62
9	45	23	99	44
11	49	78	64	29
27	58	65	37	78
8	97	33	53	19
41	75	21	61	81
19	42	29	32	26
35	55	0	50	23

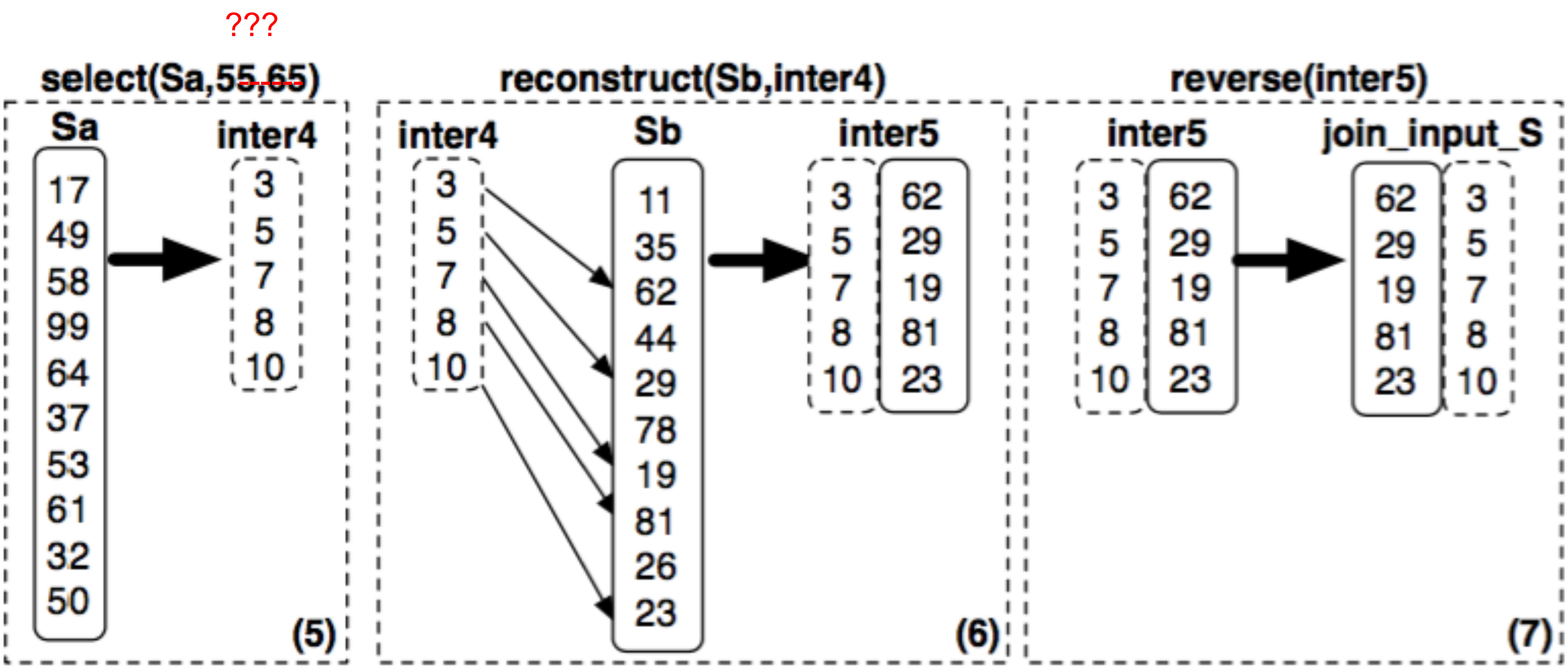
Late Materialization

select sum(R.a) from R, S
where R.c = S.b
and 5<R.a<20 and 40<R.b<50
and 30<S.a<40



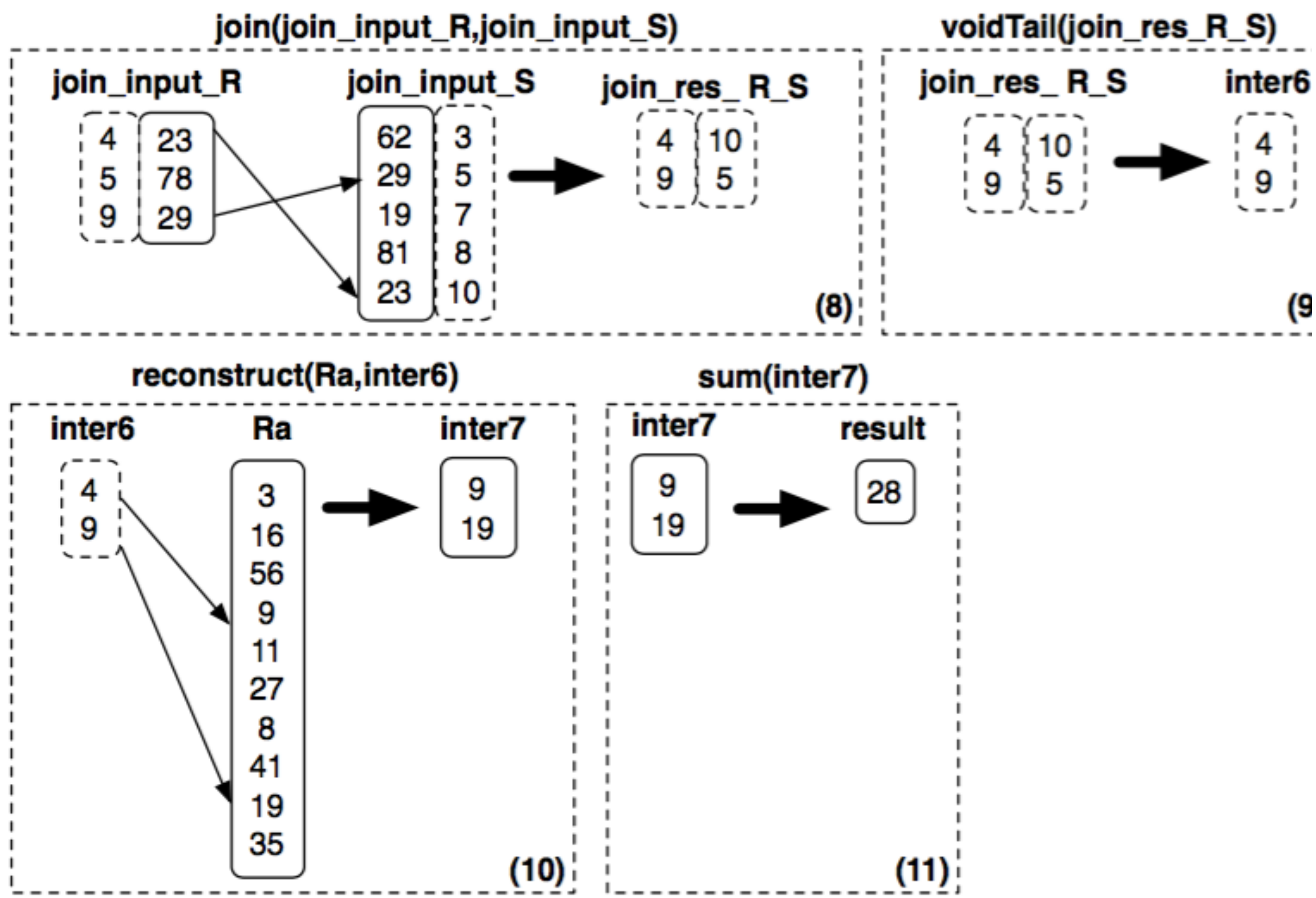
Late Materialization

```
select sum(R.a) from R, S
where R.c = S.b
 and 5<R.a<20 and 40<R.b<50
 and 30<S.a<40
```



Late Materialization

select sum(R.a) from R, S
where R.c = S.b
and 5<R.a<20 and 40<R.b<50
and 30<S.a<40



Simulating a Column-Store DBMS in a Row-Store DBMS

- Vertical partitioning
  - Two-column tables: (key, attribute)
- Index-only plans
  - Create a B+ tree index on each attribute
  - Answer queries using indexes only, without reading actual data
- Materialized views
  - Each view contains a subset of columns

Conclusion

- Column-store DBMS outperforms row-store DBMS
  - Measured on a data warehousing benchmark (SSBM)
- Late materialization and compression are key factors
- Difficult to simulate a column-store in a row-store
  - Tuple overheads cause data blow-up
  - Column joins are expensive
  - Hard to get the DBMS to “do the right thing” (e.g., index plans)
- Not the end of the story, however, … see CIDR’09 paper

Conclusion

Teaching an Old Elephant New Tricks

Nicolas Bruno
Microsoft Research
nicolasb@microsoft.com

ABSTRACT

In recent years, column stores (or C-stores for short) have emerged as a novel approach to deal with read-mostly data warehousing applications. Experimental evidence suggests that, for certain types of queries, the new features of C-stores result in orders of magnitude improvement over traditional relational engines. At the same time, some C-store proponents argue that C-stores are fundamentally different from traditional engines, and therefore their benefits cannot be incorporated into a relational engine short of a complete rewrite. In this paper we challenge this claim and show that many of the benefits of C-stores can indeed be simulated in traditional engines with no changes whatsoever. We then identify some limitations of our "pure-simulation" approach for the case of more complex queries. Finally, we predict that traditional relational engines will eventually leverage most of the benefits of C-stores natively, as is currently happening in other domains such as XML data.

CIDR'09