Column Databases

Exercise

- Tables:
  - Sales(Date,FKShop,FKCust,FKProd,UnitPrice,Q,TotPrice)
  - Shops(PKShop,Name,City,Region,State)
  - Customer(PKCust,Nome,FamName,City,Region,State,Income)
  - Products(PKProd,Name,SubCategory,Category,Price)

Exercise

- Sales: NRec: 100.000.000, Npag: 1.000.000; Shops: 500, 2; Customers: 100.000, 1.000; Products: 10.000, 100

  SELECT Sh.Region, Month(S.Date),Sum(TotPrice)

  FROM Sales S join Shops Sh on FKShops=PKShops

  GROUP BY Sh.Region, Month(S.Date)
- Propose a data organization based on this query where Month(18/05/2018)='05/2018' (not '05')
- Consider primary organization and indexes
- Consider the possibility of denormalization and vertical partitioning

Exercise

- Compute the cost of an optimal access plan based on this organization
- Add a condition: WHERE 1/1/2017 < Date
- Assume that we want to optimize some variants as well (where the SELECT clause changes according to the GROUP BY)

  …GROUP BY Sh.City, Year(S.Date)

  …GROUP BY S.Date

  …GROUP BY Sh.Region, S.FKCust

References

- The Design and Implementation of Modern Column-Oriented Database Systems, D. Abadi, P. Boncz, S. Harizopoulos, S. Idreos and S. Madden, Foundations and Trends in Databases, Vol. 5, No. 3 (2012) 197–280

Column stores in 1985

- Row store (N-ary Storage Model, NSM)

Page Header			
Id1	John	32	HK245
Id3	Mary	33	HK324
Id4	John	45	HK245

- Column store (Decomposition Storage Model, DSM)

Header	
1	Id1
2	Id3
3	Id4

Header	
1	John
2	Mary
3	John

Header	
1	32
2	33
3	45

Header	
1	HK245
2	HK324
3	HK245

The reasons behind current trend

- Applicative:
  - Diffusion of analytical tasks
- Technological:
  - Widening the RAM – I/O time gap: must access disk less
  - Widening the seek-transfer gap: disk access must be sequential
  - RAM is bigger: I/O is not the only concern any more
  - Widening the cache – RAM time gap: must reduce the cache miss
  - Instruction pipelining: must reduce function call
  - SIMD instruction: make better use of SIMD parallelism

Column stores: pro and cons

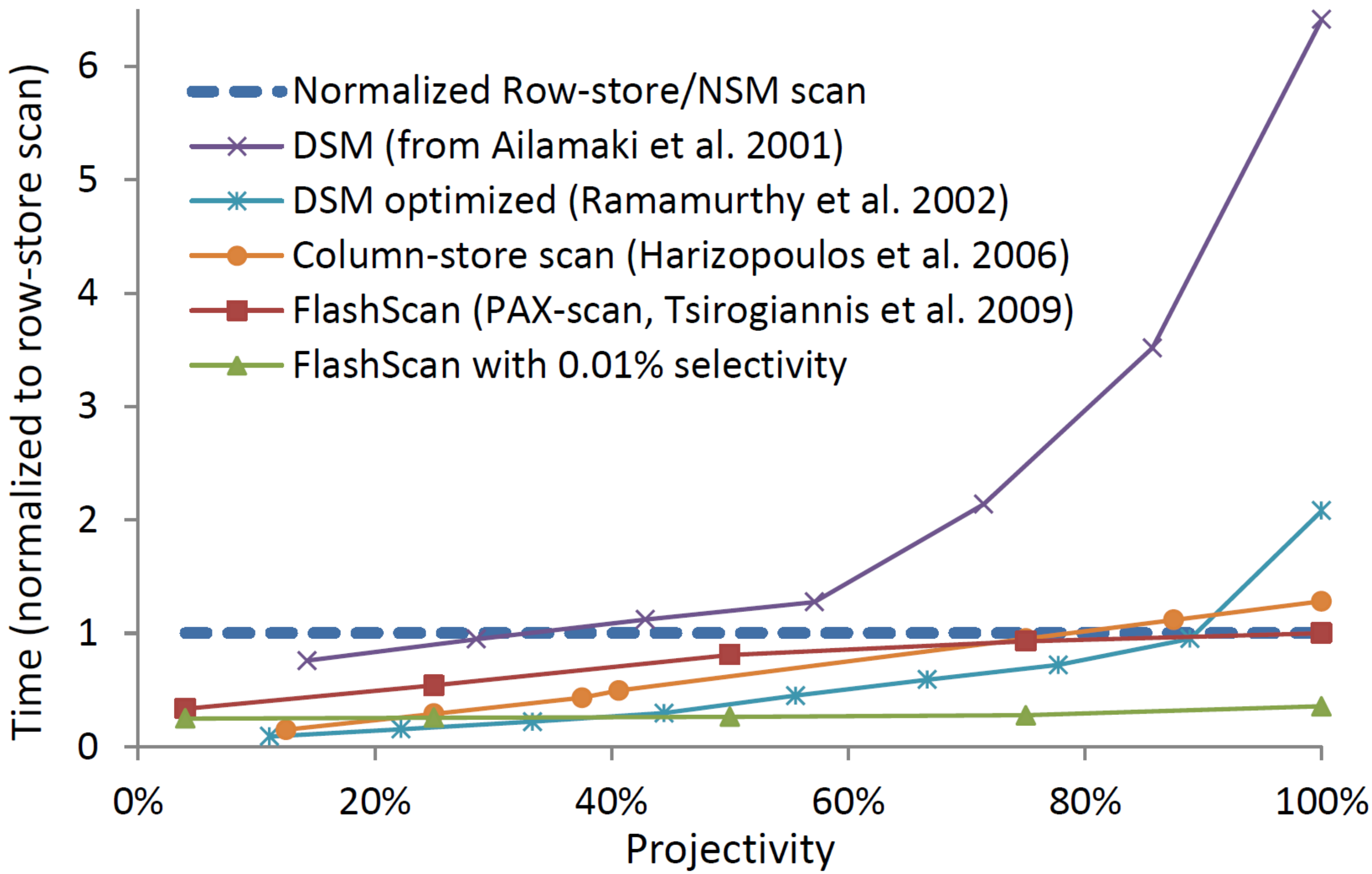
- The advantages of column stores
  - Read the relevant columns only
  - Columns are easier to compress than rows: less I/O
  - Moving to tuple-at-a-time to block-at-a-time:
    - Better use of cache, pipelining, SIMD
- The problems with column stores
  - Tuple reconstruction requires a join
  - Tuple insertion requires many I/O operations

The basic trade-off:

- Reading five fields out of one tuple of twenty fields:
  - Row store: 1 page
  - Column store: 5 pages
- Reading five fields out of a thousand consecutive tuples of twenty fields:
  - Row store: read 1000 tuples of 20 fields: 20.000 fields, 20 pages
  - Column store: read 5 columns for 1000 of values: 5.000 fields, 5 pages

Column store performance

- From [Abadi et al. 2012], like many other pictures



How to sort a column

- Columns stored in RID order:
  - Tuple reconstruction by sort-merge-like algorithm, parallel scanning
  - No need to store the RID
- Column stored in value order:
  - Excellent compression by Run-Length-Encoding
  - Range search with optimal efficiency
  - Needs to store the RID in some way
  - Tuple reconstruction by index-nested-loop like random access
  - The column looks like an index

Storing a column: the RID

Sorted by RID

RID-value

1	20	2	50
3	45	4	20

5	45	6	20
7	34	8	50

One RID per page

1	20	50	45
20	45	20	34

8	50		

Implicit RID

20	50	45	20
45	20	34	50

Sorted by value

Value-RIDS

20	1	4	6
34	7	45	3

5	50	2	8

Column + Join index

20	*3	34	*1
45	*2	50	*2

1	1	4	6
2	7	3	3

5	4	2	8

C-Store: projections

- A *projection* is a set of columns, from one table or from two or more joined tables, stored together and sorted according to one of them
- Optimal for $\sigma_{k1<date<k2}\pi_{saleid,date,region}(Sales)$

(saleid,date,region | date)

	saleid	date	region
1	17	1/6/08	West
2	22	1/6/08	East
3	6	1/8/08	South
4	98	1/13/08	South
5	12	1/20/08	North
6	4	1/24/08	South
7	14	2/2/08	West
8	7	2/4/08	North
9	8	2/5/08	East
10	11	2/12/08	East

(a) Sales Projection Sorted By Date

(prodid,date,region | region,date)

	prodid	date	region
1	5	1/6/08	East
2	9	2/5/08	East
3	4	2/12/08	East
4	12	1/20/08	North
5	5	2/4/08	North
6	7	1/8/08	South
7	22	1/13/08	South
8	3	1/24/08	South
9	18	1/6/08	West
10	6	2/2/08	West

(b) Sales Projection Sorted By Region, Date

The projections and the table

- For a given table, we store many projections, whose columns may overlap
- The usual trade-off: if we have lots of overlapping projections, many queries find their optimal projection, but updates are slower
- In order to be able to reconstruct the entire table we have two choices:
  - Having a chain of join-indexes that connect the different projections
  - Having one super-projection that contains all columns (the typical choice)

Projections

- Storing (saleid, date, region | date)
- Advantages of sorting *date* by *date*:
  - Better compression
  - Optimal I/O for range queries on *date*
  - No cost for *group-by* on *date*
- Advantages of sorting *saleid, region* by *date*:
  - No cost for join with similarly sorted columns
  - Optimal I/O for range queries on *date*
  - No cost for *group-by* on *date*

What is a projection

- It is not storing half-rows:
  - Every column is stored as a column
- A single sorted column is like a compressed index
- A projection (saleid, date, region | date) is like a compressed index on *date* with the additional attributed *saleid* and *region*
- If a column is not RID-sorted, then RIDs should be stored, either in the column or in a Join Index
- Can we avoid that?

Vectorized execution

- Traditional dicotomy:
  - Tuple-at-a-type (tuple pipeline) execution: too many function calls, does not exploit array parallelism
  - Full execution of each operator: much faster on modern CPUs but intermediate results may not fit main memory
- Solution: each ‘next()’ call returns a block of values – typically, 1000 – that fits L1 cache size
- This reduces 1000 times the number of next() call
- A tight loop on a 1000 elements array can be SIMD parallelized and subject to parallel cache load

Compression

- Trading I/O – or even memory access - vs CPU
- Bit saving means more values in the registers
- Importance of fixed-width arrays

Compression algorithms

- RLE (Run Lengh Encoding):
  - a,a,a,c,c,d,c,c,c,c,: (a,1,3), (c,4,2), (d,6,1), (c,7,4)
- Bit-Vector encoding:
  - a,b,a,a,b,c,b: a:1011000, b:0100101, c:0000010
- Dictionary encoding:
  - john,mary,john,john: (0:john)(1:mary) – 0,1,0,0

Compression algorithms

- Frame Of Reference
  - 1003,1017,1005: 1010 + (-7, +7, -5)
- Difference encoding
  - 1003,1017,1005: 1003, +14, -12
- Frequency partitioning:
  - Put similar values in the same page
  - Separate dictionary per page
- The patching technique: see the book if curious

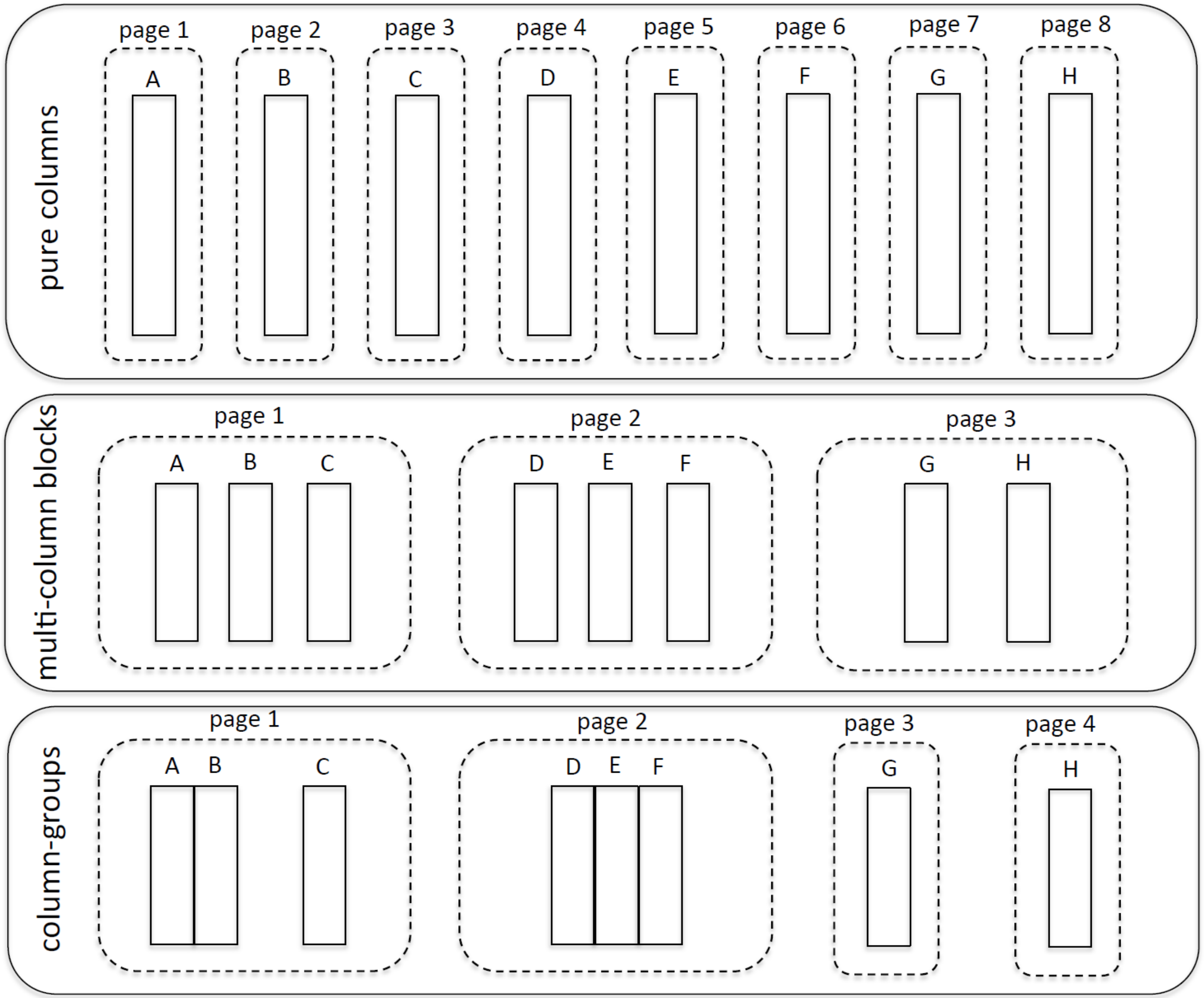
Operating on compressed data

- RLE encoded data can be operated upon without decompression
- Monotone dictionary encoding allows range queries
- Bit-Vectory encoding allows bit operations on sets

Tuple reconstruction

- Early materalization:
  - ScanTable+Project(A,B,C) -> ScanColumn(A)+ScanColumn(B)+Join+ScanColumn(C)+Join
- Late materialization: Column Algebras
- Multi-column blocks

Storage formats



Binary algebras: MIL

- XML primitives for querying a fragmented world, PA Boncz, ML Kersten, the VLDB Journal 8 (2), 101-119
- Every table is binary (or unary as a special case)
- **select**(bat[H,T] **AB**,bool ***f**,…pi…) : bat[H,nil]

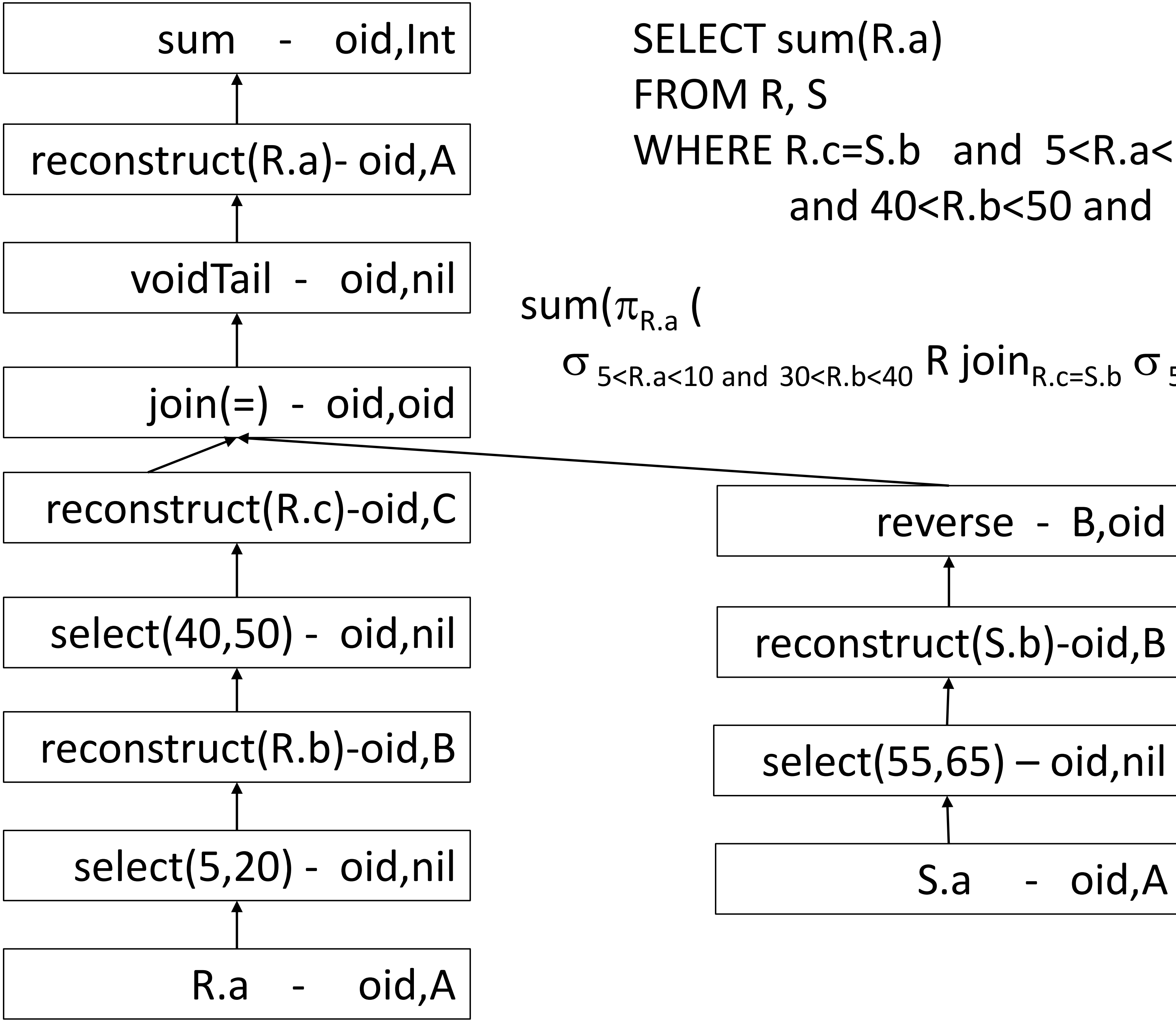
  $= \langle$ [a,nil] | [a,b]$\in$AB $\wedge$ (*f)(b,…pi…) $\rangle$
- **join**(bat[T1,T2] **AB**, bat[T2,T3] **CD**, bool ***f**,…pi…) : bat[T1,T3]

  $= \langle$ [a,d] | [a,b]$\in$AB $\wedge$ [c,d]$\in$CD $\wedge$ (*f)(b,c,…pi…) $\rangle$
- **reconstruct**(bat[H,nil] **AN**, bat[H,T] **AB**) : bat[H,T]

  $= \langle$ [a,b] | [a,b]$\in$AB $\wedge$ [a,nil]$\in$AN $\rangle$

Binary algebras

- **reverse**(bat[H,T] AB): bat[T,H]
- $= \langle$ [b,a] | [a,b]$\in$AB $\rangle$
- **voidtail**(bat[H,T] AB): bat[H,nil] $= \langle$ [a,nil] | [a,b]$\in$AB $\rangle$
- *group*{[a,A],[b,B],[c,A],[d,B]} = {[a,a],[b,b],[c,a],[d,b]}
- **group**(bat[oid,T] AB): bat[oid,oid]
  = { [a,o] | o = id_{AB}(b) $\wedge$ [a,b]$\in$AB }
  where id_{AB} is a bijection and [id_{AB}(x),x] $\in$AB
- **sum**(bat[oid,Int] AB): bat[oid,Int]
  = [nil,sum{i | [o,i] $\in$AB }]



SELECT sum(R.a)
FROM R, S
WHERE R.c=S.b and 5<R.a<20
and 40<R.b<50 and 55<S.a<65

$\text{sum}(\pi_{R.a}($
$\sigma_{5<R.a<10 \text{ and } 30<R.b<40}\ R\ \text{join}_{R.c=S.b}\ \sigma_{55<S.a<65}\ S))$

$$\text{sum}(\pi_{R.a}\ (\sigma_{5<R.a<20 \text{ and } 40<R.b<50}\ R\ \text{join}_{R.c=S.b}\ \sigma_{55<S.a<65}\ S))$$

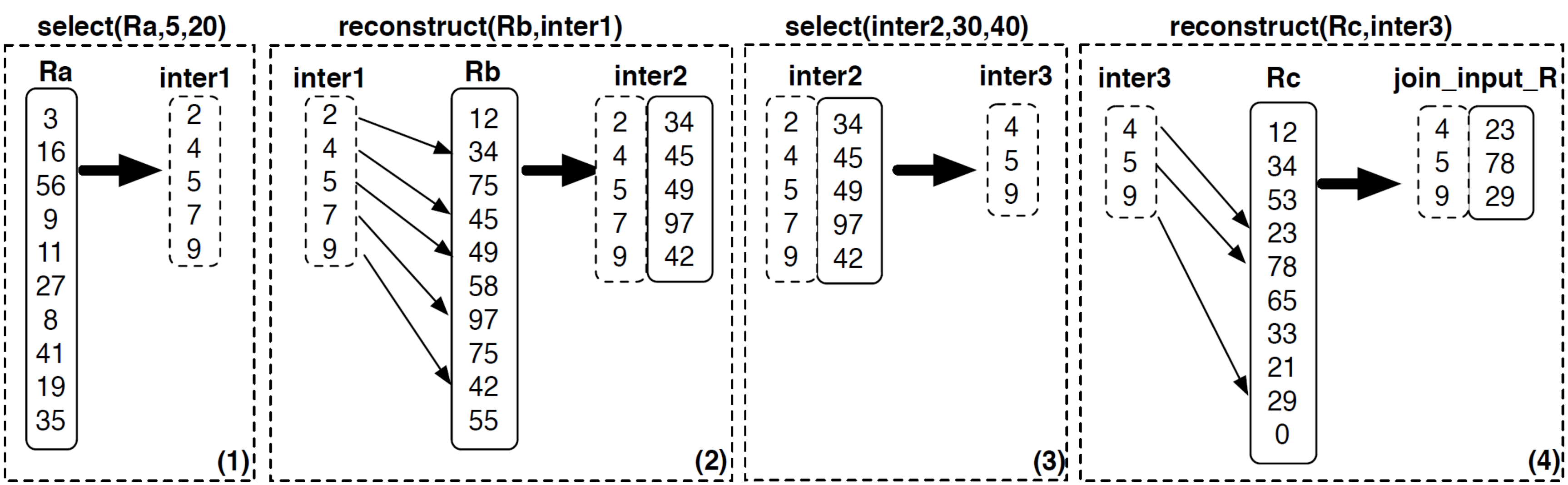
Initial Status

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

Query and Query Plan (MAL Algebra)

SELECT sum(R.a) FROM R, S WHERE R.c=S.b
and 5<R.a<20 and 40<R.b<50 and 55<S.a<65

1. inter1 = **select**(Ra,5,20)
2. inter2 = **reconstruct**(Rb,inter1)
3. inter3 = **select**(inter2,40,50)
4. join_input_R = **reconstruct**(Rc,inter3)
5. inter4 = **select**(Sa,55,65)
6. inter5 = **reconstruct**(Sb,inter4)
7. join_input_S = **reverse**(inter5)
8. join_res_R_S = **join**(join_input_R,join_input_S)
9. inter6 = **voidTail**(join_res_R_S)
10. inter7 = **reconstruct**(Ra,inter6)
11. result = **sum**(inter7)



Inter1 = $\sigma_{5<R.a<20}$ Ra

Inter2 = Rb semijoin $\sigma_{5<R.a<20}$ Ra

Inter3 = $\sigma_{40<R.b<50}$ Inter2

join_input_R = Rc semijoin Inter3

$$\mathrm{sum}(\pi_{R.a}\,(\sigma_{5<R.a<20 \text{ and } 40<R.b<50}\ R\ \mathrm{join}_{R.c=S.b}\ \sigma_{55<S.a<65}\ S))$$

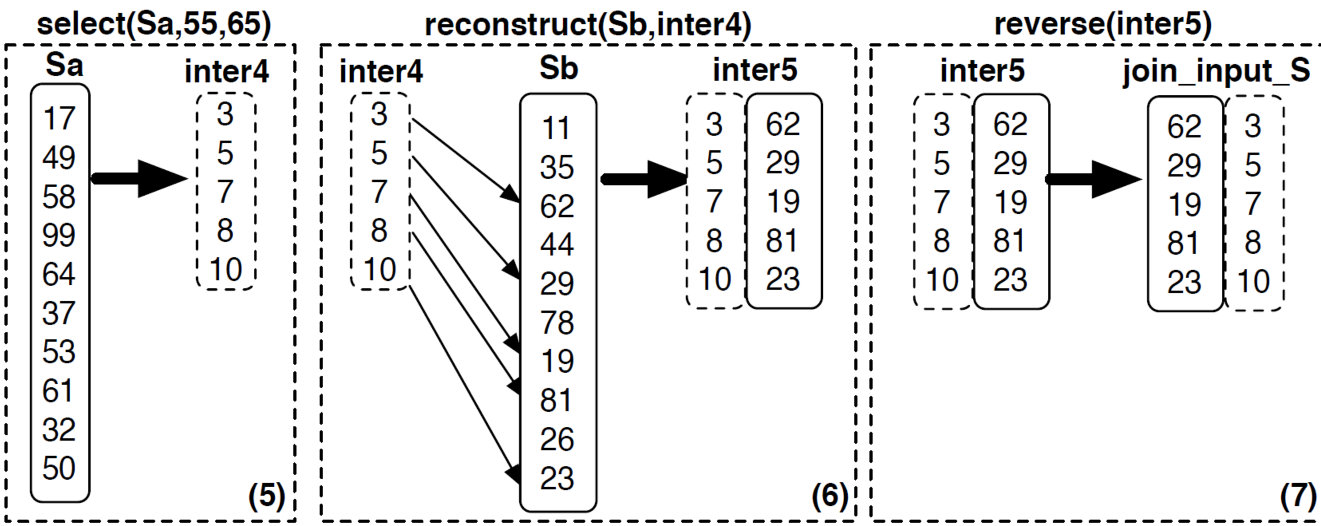
Initial Status

Relation R: Ra	Rb	Rc	Relation S: 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

Query and Query Plan (MAL Algebra)

SELECT sum(R.a) FROM R, S WHERE R.c=S.b
and 5<R.a<20 and 40<R.b<50 and 55<S.a<65

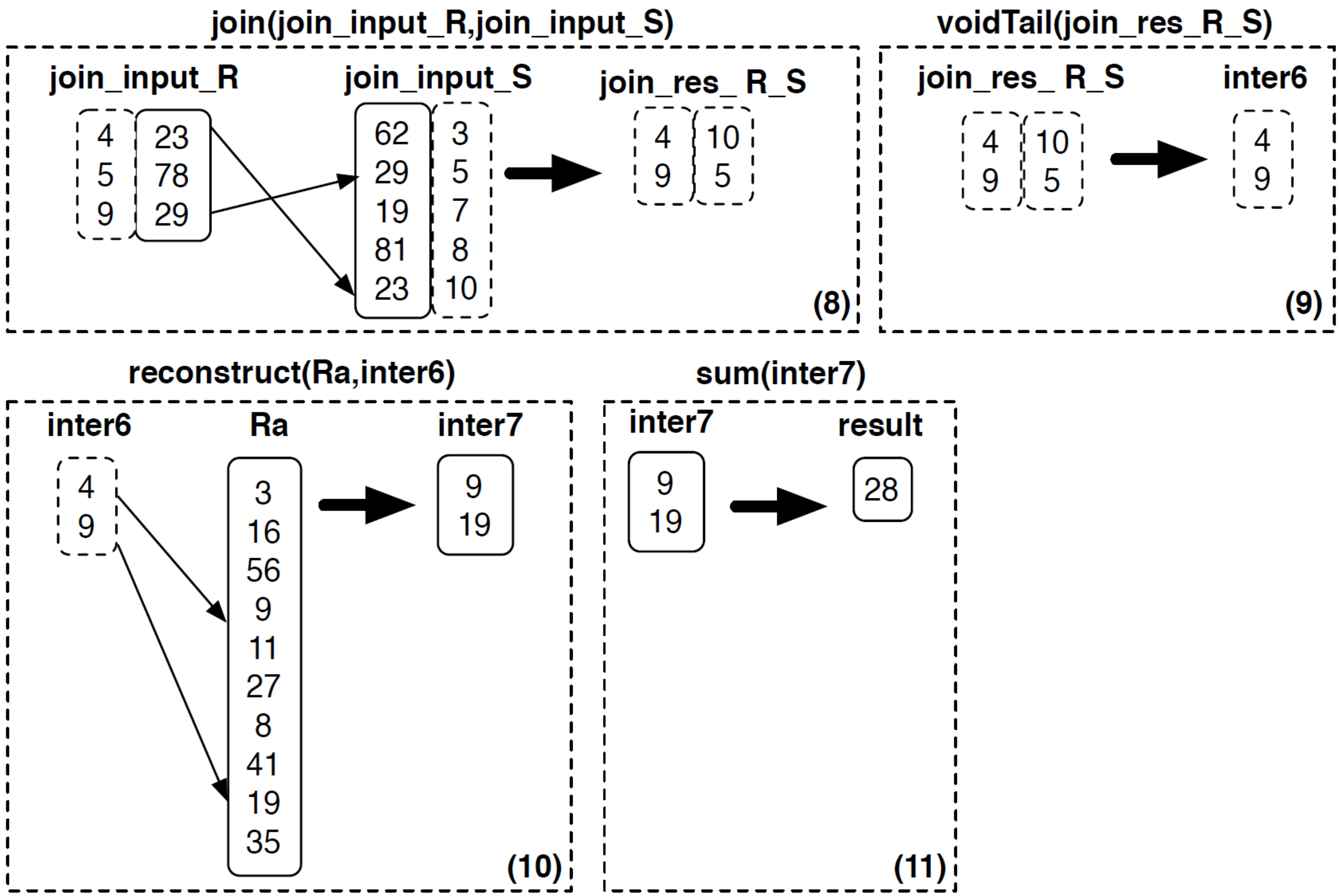
1. inter1 = **select**(Ra,5,20)
2. inter2 = **reconstruct**(Rb,inter1)
3. inter3 = **select**(inter2,40,50)
4. join_input_R = **reconstruct**(Rc,inter3)
5. inter4 = **select**(Sa,55,65)
6. inter5 = **reconstruct**(Sb,inter4)
7. join_input_S = **reverse**(inter5)
8. join_res_R_S = **join**(join_input_R,join_input_S)
9. inter6 = **voidTail**(join_res_R_S)
10. inter7 = **reconstruct**(Ra,inter6)
11. result = **sum**(inter7)



$\mathrm{Inter4} = \sigma_{55<S.a<65}\ \mathrm{Sa}$ $\qquad$ Inter5 = Rb semijoin Inter4

$\mathrm{join\_input\_S} = \pi_{S.b}\ \sigma_{55<S.a<65}\ S$

$$sum(\pi_{R.a}\ (\sigma_{\ 5<R.a<20 \text{ and } 40<R.b<50}\ R\ join_{R.c=S.b}\ \sigma_{\ 55<S.a<65}\ S))$$



8. $join\_res = (\pi_{R.c}\ \sigma_{\ 5<R.a<20 \text{ and } 40<R.b<50}\ R)\ join\ (\pi_{S.b}\ \sigma_{\ 55<S.a<65}\ S)$

9. $Inter6 = \pi_{R.*}\ (\sigma_{\ 5<R.a<20 \text{ and } 40<R.b<50}\ R\ join_{R.c=S.b}\ \sigma_{\ 55<S.a<65}\ S)$

10. $Inter7 = \pi_{R.a}\ (\sigma_{\ 5<R.a<20 \text{ and } 40<R.b<50}\ R\ join_{R.c=S.b}\ \sigma_{\ 55<S.a<65}\ S)$

11. $Result = sum(\pi_{R.a}\ (\sigma_{\ 5<R.a<20 \text{ and } 40<R.b<50}\ R\ join_{R.c=S.b}\ \sigma_{\ 55<S.a<65}\ S))$

Representing a BAT

- A BAT where the first column is a list of consecutive oids will be represented as its range plus the second column:

7	87
8	89
9	89
10	89

(7,10)

87
89
89
89

(7,10)RLE

87,1
89,3

Representing a BAT

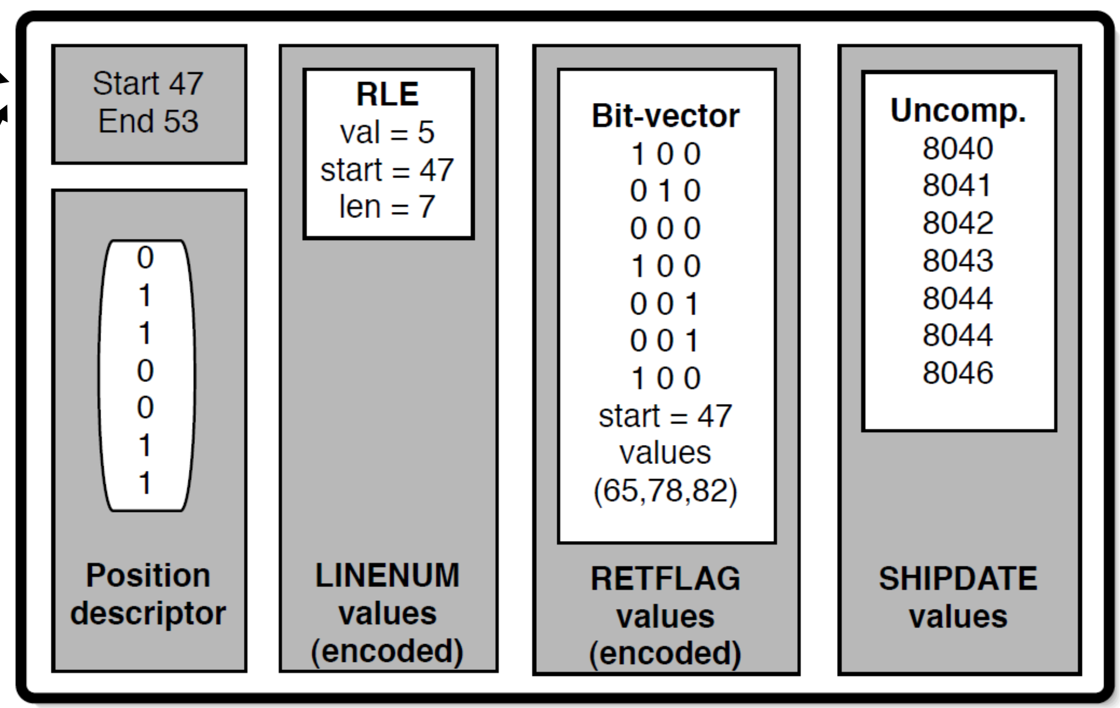
- The result of a select may just be a bitmap
- The result of a reconstruct may just be a bitmap in front of the original column

Shipdate

48	8041
49	8042
52	8044
53	8046

Linenum

48	5
49	5
52	5
53	5

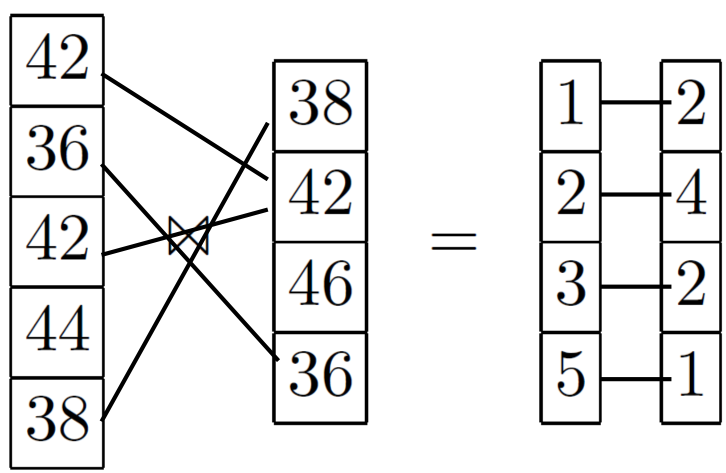


Exercise

- How would you execute select(30,35) over a integer sorted column compressed as follows, assuming to represent the result as a bitmap:
  - RLE
  - Bitmap
  - Dictionary
- The resulting bitmap is compressed?
- What if the column is not sorted?
- What if the column is filtered by a bitmap?

Column join

- Column stores avoid indexes, hence:
  - Hash Join
  - Sort-merge
  - Main-memory join (columnn often fit main memory)
- Join Index: value columns -> position pairs



The Jive join

- *Fast joins using join indices*, Li, Z. & Ross, K. The VLDB Journal (1999) 8: 1, pp. 1-24.
- Main features:
  - Operator-at-a-time algorithm (pipelined version exists)
  - Single read through input relations
  - Only blocks where one tuple that is named in the join index is present are read
  - The result is in column form

The Jive join: using the join index

Student	Course
Smith[1]	101
Smith[2]	109
Jones	104
Davis[1]	102
Davis[2]	105
Davis[3]	106
Brown	102
Black	103
Frick	107

Relation *Student*

Course	Instructor
101	Green
102	Yellow
103	Green
104	White
105	Evans
106	Alberts
106	Beige
108	Red
109	Grey

Relation *Course*

Student	Course	Instructor
Smith[1]	101	Green
Smith[2]	109	Grey
Jones	104	White
Davis[1]	102	Yellow
Davis[2]	105	Evans
Davis[3]	106	Alberts
Davis[3]	106	Beige
Brown	102	Yellow
Black	103	Green

Join Result

Student tuple id	Course tuple id
1	1
2	9
3	4
4	2
5	5
6	6
6	7
7	2
8	3

Join Index

(a) id< 3	
Smith[1]	1
Davis[1]	2
Brown	2
JR_1(a)	Temp(a)

(b) 3 ≤id< 6	
Jones	4
Davis[2]	5
Black	3
JR_1(b)	Temp(b)

(c) 6 ≤id	
Smith[2]	9
Davis[3]	6
Davis[3]	7
JR_1(c)	Temp(c)

(a) id< 3		
1	1	101 Green
2	2	102 Yellow
2		

⇓

101 Green
102 Yellow
102 Yellow
JR_2(a)

(b) 3 ≤id< 6		
4	3	103 Green
5	4	104 White
3	5	105 Evans

⇓

104 White
105 Evans
103 Green
JR_2(b)

(c) 6 ≤id		
9	6	106 Alberts
6	7	106 Beige
7	9	109 Grey

⇓

109 Grey
106 Alberts
106 Beige
JR_2(c)

The Jive join

(a) id< 3	
Smith[1]	1
Davis[1]	2
Brown	2
JR_1(a)	Temp(a)

(b) 3 ≤id< 6	
Jones	4
Davis[2]	5
Black	3
JR_1(b)	Temp(b)

(c) 6 ≤id	
Smith[2]	9
Davis[3]	6
Davis[3]	7
JR_1(c)	Temp(c)

(a) id< 3		
1	1	101 Green
2	2	102 Yellow
2		

⇓

101 Green
102 Yellow
102 Yellow
JR_2(a)

(b) 3 ≤id< 6		
4	3	103 Green
5	4	104 White
3	5	105 Evans

⇓

104 White
105 Evans
103 Green
JR_2(b)

(c) 6 ≤id		
9	6	106 Alberts
6	7	106 Beige
7	9	109 Grey

⇓

109 Grey
106 Alberts
106 Beige
JR_2(c)

	Student	Course	Instructor	
	Smith[1]	101	Green	
JR_1(a)	Davis [1]	102	Yellow	JR_2(a)
	Brown	102	Yellow	
	Jones	104	White	
JR_1(b)	Davis[2]	105	Evans	JR_2(b)
	Black	103	Green	
	Smith[2]	109	Grey	
JR_1(c)	Davis[3]	106	Alberts	JR_2(c)
	Davis[3]	106	Beige	

The Jive join: assumptions

- R(RId,A,B,…) and S(SId,BB,C,…)
- Want to compute π_{ABC} (R $\bowtie_{B=BB}$ S)
- We have the JoinIndex JI = $\pi_{RId,SId}$ (R $\bowtie_{B=BB}$ S)
- We have 2*NPag < B*B, but NPag only includes pages in the projected (and semijoined) S plus the pages of the Join Index:
  – 2*(NPag (π_{CD} (R $\bowtie_{B=BB}$ S)) + Npag(JI)) < B*B
- No limit on R

The Jive join

- We partition projected $\pi_{SId,BB,C}$ S in K files S_i such that each partition fits main memory (but K<B)
- Partition is logical!
- We scan R and JI in parallel. We use 2*K buffers to create 2*K files that contain:
  - $JR_i = \pi_{ARid} (R \bowtie_{B=BB} S_i)$ (the K output files)
  - $JSId_i = \pi_{SId} (R \bowtie_{B=BB} S_i)$ (the K temporary files)
- We are almost done: we must just create, for each JR_i, a correponding JS_i that contains $\pi_{BBC} (R \bowtie_{B=BB} S_i)$ in the same order

The Jive join

- For each i, we load the whole $JSId_i$ in memory, sort a copy in SId order, scan the corresponding i-partition of $\pi_{SId,BB,C}$ S, skipping the useless blocks – call it S_i
- We scan $JSId_i$. For each SId in $JSId_i$, we put the corresponding record of S_i (which is in memory) in JS_i . Now, JS_i containes the joined records in the same order as $JSId_i$ and JR_i
- Hence we have:
  - $JR_i = \pi_A (R \bowtie_{B=BB} S_i)$
  - $JS_i = \pi_{BBC} (R \bowtie_{B=BB} S_i)$

The cost

- First phase:
  - Read R (only the block that have joined tuples) and JI. Write π_{SId} (R $\bowtie_{B=BB}$ S) and π_{A} (R $\bowtie_{B=BB}$ S).
- Second phase:
  - Read S (only the block that have joined tuples) and π_{SId} (R $\bowtie_{B=BB}$ S) . Write π_{BBC} (R $\bowtie_{B=BB}$ S)
- In short: read R and S once (maybe skipping some blocks), read JI, write and re-read π_{SId} (R $\bowtie_{B=BB}$ S), write the result

Memory needs

- Phase 1: we create 2*K files, we use a buffer of length l for each file, hence K = B/(2*l)
- Phase 2:
  - Size of JSIdi: |JI|/K
  - Plus size of $\pi_{SId,BB,C}$ S – SId may be implicit: $|\pi S|/K$
- Hence
  - $(|JI|+|\pi S|)/K < B \rightarrow K = (|JI|+|\pi S|)/B$
  - $2*l*(|JI|+|\pi S|) < B*B \rightarrow l = (B*B)/(2*(|JI|+|\pi S|))$
- 'l' depends on the ratio between B*B and S
- No limit on the size of R

Group by and aggregation

- Group by is typically hash-based, unless, of course, the input is already sorted according to the group-by attributes
- Aggregation (sum, count…) takes advantage of columnar layout in order to perform a tight loop over a cache-sized array

Insert / update / delete

- Insertion is expensive:
  - One insert for each column
  - Redundant representation implies redundant insertions
- If a column is ordered, then insertion must respect the order
- Decompression / recompression is often needed
- Updates and deletes have the same problems
- Solution: differential files

Read Store and Write Store

- Mature data live in the big Read Store and fresh Updates (insertions/deletions/updates) are in the small main-memory Write Store
- Read Store is read-optimized while Write Store is compact
- Every query queries both the RS and the WS and merges the two results – the WS results contains insertions to be added and deletions to be removed
- Problem: this approach requires an explicit RID

RS / WS and transactions

- Implementing Snapshot Isolation:
  - The RS is the Snapshot
  - Each transaction has its own WS
- Implementing no-undo / no-log concurrency control:
  - The WS is merged only after commit
  - The WS is the redo log

Indexing in column stores

- A sorted column A with a join index corresponds to a traditional index A
- A projection { A, B, C | C } may be considered as an index on C that allows one to rapidly access A and B, with an IndexOnly plan

Simulating columns with indexes

- We create an index on every column of R(A,B,C,D,…)
- Would typically not work:
  - Tuple reconstruction requires a Join of the index with the original table
  - Insert/delete overhead

Simulating columns: vertical partitioning

- We may rewrite the table R(A,B,C,D,…) as a set of tables R(IdA,A), R(IdA,B), R(IdA,C), where IdA is a small integer, and the tables are IdA-sorted for a fast join.
- However:
  - No compression
  - IdA overhead
  - Pipelined execution
  - Lots of joins may confuse the optimizer
  - Insert/delete overhead

Conclusions: a new landscape

- Applications: OLAP vs OLTP applications, ingesting whole tables rather than looking for a tuple
- Main memory is getting bigger and bigger
- Disk seek time is getting relatively slower
- Memory access is getting slower: needs to exploit data and code cache
- On-chip parallelism must be exploited: SIMD operations and pipelined executions: more arrays, less function calls

Conclusions: is not just vertical partitioning

- Only read the columns that you need
- Keep the columns compressed
- Store redundant projections in many orders
- Never rebuild the tuple until the end
- Use fixed-size columns and work on a cache-sized array at a time

Exercise: the cost of the following plan

```sql
SELECT sum(R.a)
FROM R, S
WHERE R.c=S.b   and  5<R.a<10
          and 30<R.b<40 and  55<S.a<65
```

$$\mathrm{sum}(\pi_{R.a}(\sigma_{5<R.a<10 \text{ and } 30<R.b<40}\, R \ \mathrm{join}_{R.c=S.b}\ \sigma_{55<S.a<65}\, S))$$