Column Databases
Exercise

• Tables:
  – Sales(Date, FKShop, FKCust, FKProd, UnitPrice, Q, TotPrice)
  – Shops(PKShop, Name, City, Region, State)
  – Customer(PK Cust, 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

```sql
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

Column stores in 1985

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

<table>
<thead>
<tr>
<th>Id1</th>
<th>John</th>
<th>32</th>
<th>HK245</th>
</tr>
</thead>
<tbody>
<tr>
<td>Id3</td>
<td>Mary</td>
<td>33</td>
<td>HK324</td>
</tr>
<tr>
<td>Id4</td>
<td>John</td>
<td>45</td>
<td>HK245</td>
</tr>
</tbody>
</table>

- Column store (Decomposition Storage Model, DSM)

<table>
<thead>
<tr>
<th>Header</th>
<th>Id1</th>
<th>John</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Id3</td>
<td>Mary</td>
<td>33</td>
</tr>
<tr>
<td>3</td>
<td>Id4</td>
<td>John</td>
<td>45</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Header</th>
<th>1</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>HK245</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>HK324</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>HK245</td>
</tr>
</tbody>
</table>
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
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

<table>
<thead>
<tr>
<th>Sorted by RID</th>
<th>RID-value</th>
<th>One RID per page</th>
<th>Implicit RID</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1 20 2 50</td>
<td>1 20 50 45</td>
<td>20 50 45 20</td>
</tr>
<tr>
<td></td>
<td>3 45 4 20</td>
<td>20 45 20 34</td>
<td>45 20 34 50</td>
</tr>
<tr>
<td></td>
<td>5 45 6 20</td>
<td>8 50</td>
<td></td>
</tr>
<tr>
<td></td>
<td>7 34 8 50</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Sorted by value</th>
<th>Value-RIDS</th>
<th>Column + Join index</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>20 1 4 6</td>
<td>20 *3 34 *1</td>
</tr>
<tr>
<td></td>
<td>34 7 45 3</td>
<td>45 *2 50 *2</td>
</tr>
<tr>
<td></td>
<td>5 50 2 8</td>
<td></td>
</tr>
</tbody>
</table>

*Note: The values in the table represent RIDs or values.*

- **Sorted by RID:** The RIDs are sorted in ascending order. The RIDs 1, 20, 2, 50 are sorted in this manner.
- **Sorted by value:** The values are sorted in ascending order. The values 20, 1, 4, 6 are sorted in this manner.
- **One RID per page:** Each page contains a unique RID. The RIDs 1, 20, 50, 45 are used here.
- **Implicit RID:** This method does not explicitly store RIDs. It uses values directly for indexing.
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 $\sigma_{k_1 < date < k_2} \pi_{\text{saleid, date, region}} (\text{Sales})$
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 Length 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 materialization:
  – 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)
- \textbf{select}(\text{bat}[H,T] \ AB, \text{bool } *f, \ldots \pi \ldots) : \text{bat}[H,nil] = \langle [a,nil] | [a,b] \in AB \land (\ast f)(b, \ldots \pi \ldots) \rangle
- \textbf{join}(\text{bat}[T1,T2] \ AB, \text{bat}[T2,T3] \ CD, \text{bool } *f, \ldots \pi \ldots) : \text{bat}[T1,T3] = \langle [a,d] | [a,b] \in AB \land [c,d] \in AB \land (\ast f)(b,c, \ldots \pi \ldots) \rangle
- \textbf{reconstruct}(\text{bat}[H,nil] \ AN, \text{bat}[H,T] \ BC) : \text{bat}[H,T] = \langle [a,b] | [a,b] \in AB \land [a,nil] \in AN \rangle
Binary algebras

• **reverse**(bat[H,T] AB): bat[T,H]
  
  = ⟨ [b,a] | [a,b] ∈ AB ⟩

• **voidtail**(bat[H,T] AB): bat[H,nil] = ⟨ [a,nil] | [a,b] ∈ AB ⟩

• **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) ∧ [a,b] ∈ AB }
  where id_{AB} is a bijection and [id_{AB}(x),x] ∈ AB

• **sum**(bat[oid,Int] AB): bat[oid,Int]
  = [nil,sum{i | [o,i] ∈ AB }]
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
Inter1 = $\sigma_{5<R.a<10} R_a$

Inter3 = $\sigma_{30<R.b<40} \text{Inter2}$

join_input_R = $R_c \text{ semijoin } \text{Inter3}$

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

**Initial Status**

<table>
<thead>
<tr>
<th>Ra</th>
<th>Rb</th>
<th>Rc</th>
<th>Sa</th>
<th>Sb</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>12</td>
<td>12</td>
<td>17</td>
<td>11</td>
</tr>
<tr>
<td>16</td>
<td>34</td>
<td>34</td>
<td>49</td>
<td>35</td>
</tr>
<tr>
<td>56</td>
<td>75</td>
<td>53</td>
<td>58</td>
<td>62</td>
</tr>
<tr>
<td>9</td>
<td>45</td>
<td>23</td>
<td>99</td>
<td>44</td>
</tr>
<tr>
<td>11</td>
<td>49</td>
<td>78</td>
<td>64</td>
<td>29</td>
</tr>
<tr>
<td>27</td>
<td>58</td>
<td>65</td>
<td>37</td>
<td>78</td>
</tr>
<tr>
<td>8</td>
<td>97</td>
<td>33</td>
<td>53</td>
<td>19</td>
</tr>
<tr>
<td>41</td>
<td>21</td>
<td>61</td>
<td>81</td>
<td>32</td>
</tr>
<tr>
<td>19</td>
<td>29</td>
<td>26</td>
<td>50</td>
<td>23</td>
</tr>
<tr>
<td>35</td>
<td>55</td>
<td>0</td>
<td>50</td>
<td>23</td>
</tr>
</tbody>
</table>

**Query and Query Plan (MAL Algebra)**

1. inter1 = select(Ra, 5, 20)
2. inter2 = reconstruct(Rb, inter1)
3. inter3 = select(inter2, 30, 40)
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)

**Inter4** = \( \sigma_{55<S.a<65} \) \( Sa \)  

**Inter5** = \( Rb \) semijoin \( \text{Inter4} \)  

**join_input_S** = \( \pi_{S.b} \ (\sigma_{55<S.a<65} S) \)
8. \( \text{join\_res} = (\pi_{R.c} \quad \sigma_{5 < R.a < 10 \text{ and } 30 < R.b < 40} \quad R \quad \text{join} \quad \pi_{S.b} \quad \sigma_{55 < S.a < 65} \quad S) \) \\
9. \( \text{Inter6} = \pi_{R.*} \quad (\sigma_{5 < R.a < 10 \text{ and } 30 < R.b < 40} \quad R \quad \text{join} \quad R.c = S.b \quad \sigma_{55 < S.a < 65} \quad S) \) \\
10. \( \text{Inter7} = \pi_{R.a} \quad (\sigma_{5 < R.a < 10 \text{ and } 30 < R.b < 40} \quad R \quad \text{join} \quad R.c = S.b \quad \sigma_{55 < S.a < 65} \quad S) \) \\
11. \( \text{Result} = \sum(\pi_{R.a} \quad (\sigma_{5 < R.a < 10 \text{ and } 30 < R.b < 40} \quad R \quad \text{join} \quad R.c = S.b \quad \sigma_{55 < S.a < 65} \quad S)) \)
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:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>87</td>
</tr>
<tr>
<td>8</td>
<td>89</td>
</tr>
<tr>
<td>9</td>
<td>89</td>
</tr>
<tr>
<td>10</td>
<td>89</td>
</tr>
</tbody>
</table>

(7,10) 87

(7,10)RLE 87,1

89

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
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 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

The Jive join: using the join index

<table>
<thead>
<tr>
<th>Student</th>
<th>Course</th>
<th>Instructor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Smith¹</td>
<td>101</td>
<td>Green</td>
</tr>
<tr>
<td>Smith²</td>
<td>109</td>
<td>Yellow</td>
</tr>
<tr>
<td>Jones</td>
<td>104</td>
<td></td>
</tr>
<tr>
<td>Davis¹</td>
<td>102</td>
<td>White</td>
</tr>
<tr>
<td>Davis²</td>
<td>105</td>
<td>Evans</td>
</tr>
<tr>
<td>Davis³</td>
<td>106</td>
<td>Alberts</td>
</tr>
<tr>
<td>Brown</td>
<td>102</td>
<td>Beige</td>
</tr>
<tr>
<td>Black</td>
<td>103</td>
<td>Red</td>
</tr>
<tr>
<td>Frick</td>
<td>107</td>
<td>Grey</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Course</th>
<th>Instructor</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>Green</td>
</tr>
<tr>
<td>103</td>
<td>Green</td>
</tr>
<tr>
<td>104</td>
<td>White</td>
</tr>
<tr>
<td>105</td>
<td>Evans</td>
</tr>
<tr>
<td>106</td>
<td>Alberts</td>
</tr>
<tr>
<td>108</td>
<td>Red</td>
</tr>
<tr>
<td>109</td>
<td>Grey</td>
</tr>
</tbody>
</table>

Relation **Student**

Relation **Course**

Join Result

Join Index
### The Jive Join

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>(a) $id &lt; 3$</td>
<td>(b) $3 \leq id &lt; 6$</td>
<td>(c) $6 \leq id$</td>
</tr>
<tr>
<td>Smith$^1$</td>
<td>1</td>
<td>Smith$^2$</td>
</tr>
<tr>
<td>Davis$^1$</td>
<td>2</td>
<td>Davis$^3$</td>
</tr>
<tr>
<td>Brown</td>
<td>2</td>
<td>Davis$^3$</td>
</tr>
<tr>
<td>$JR_1(a)$</td>
<td>Temp(a)</td>
<td>$JR_1(b)$</td>
</tr>
</tbody>
</table>

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>(a) $id &lt; 3$</td>
<td>(b) $3 \leq id &lt; 6$</td>
<td>(c) $6 \leq id$</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>101 Green</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>102 Yellow</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>103 Green</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>104 White</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>105 Evans</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>106 Beige</td>
</tr>
<tr>
<td>7</td>
<td>9</td>
<td>109 Grey</td>
</tr>
<tr>
<td>$JR_2(a)$</td>
<td></td>
<td>$JR_2(b)$</td>
</tr>
</tbody>
</table>

### Table: Student-Course-Instructor Join

<table>
<thead>
<tr>
<th>Student</th>
<th>Course</th>
<th>Instructor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Smith$^1$</td>
<td>101</td>
<td>Green</td>
</tr>
<tr>
<td>Davis$^1$</td>
<td>102</td>
<td>Yellow</td>
</tr>
<tr>
<td>Brown</td>
<td>102</td>
<td>Yellow</td>
</tr>
<tr>
<td>Jones</td>
<td>104</td>
<td>White</td>
</tr>
<tr>
<td>Davis$^2$</td>
<td>105</td>
<td>Evans</td>
</tr>
<tr>
<td>Black</td>
<td>103</td>
<td>Green</td>
</tr>
<tr>
<td>Smith$^2$</td>
<td>109</td>
<td>Grey</td>
</tr>
<tr>
<td>Davis$^3$</td>
<td>106</td>
<td>Alberts</td>
</tr>
<tr>
<td>Davis$^3$</td>
<td>106</td>
<td>Beige</td>
</tr>
<tr>
<td>$JR_1(a)$</td>
<td>$JR_2(a)$</td>
<td>$JR_2(b)$</td>
</tr>
</tbody>
</table>
The Jive join: assumptions

- \( R(RId,A,B,...) \) and \( S(SId,C,D,...) \)
- Want to compute \( \pi_{ABCD} (R \; V_{A=C} \; S) \)
- We have the JoinIndex \( JI = \pi_{RId,Sd} (R \; V_{A=C} \; S) \)
- We have \( 2*NPag < B*B \), but \( NPag \) only includes pages in the projected (and semijoined) \( S \) plus the pages of the Join Index:
  - \( 2*(NPag (\pi_{CD} (R \; V_{A=C} \; S)) + Npag(JI)) < B*B \)
The Jive join

• We partition projected $\pi_{S_{\text{Id,C,D}}} S$ in $K$ files $S_i$ ($K = B/2$)
• Partition is logical!
• We scan $R$ and $JI$ in parallel. We use $2*K$ buffers to create $2*K$ files that contain:
  - $JR_i = \pi_{AB} (R \, V_{A=C} \, S_i)$ (the $K$ output files)
  - $JS_{\text{Id}} = \pi_{S_{\text{Id}}} (R \, V_{A=C} \, S_i)$ (the $K$ temporary files)
• We are almost done: we must just create, for each $JR_i$, a corresponding $JS_i$ that contains $\pi_{CD} (R \, V_{A=C} \, S_i)$ in the same order
• Cost of the phase?
The Jive join

• For each i, we load the whole JSId_i in memory, and sort a copy in SId order
• We read the corresponding i-partition of \( \pi_{S\text{Id},C,D} S \), skipping the useless blocks
• We scan the unsorted copy of JSId. For each Sid in JSId, we put the corresponding record of S_i (which is in memory) in JS_i
• Now we have:

  \[ \text{JR}_i = \pi_{AB} (R \ V_{A=C} S_i) \]

  \[ \text{JS}_i = \pi_{CD} (R \ V_{A=C} S_i) \]
Memory needs

- Phase 1: 2*K buffers, hence K = B/2 (y = m/2)
- Phase 2:
  - Size of JSIdi: |JI|/K
  - Plus size of $\pi_{S_{ld,c,d}}$ S – Sld may be implicit: |$\pi S$|/K
- Hence
  - $(|JI|+|\pi S|)/K < B$
  - $2*|JSIdi|+2*|\pi S| < B*B$
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 $\{ A, B, C \mid C \}$ 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
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

sum(π_{R.a} (σ_{5<R.a<10 and 30<R.b<40} R join_{R.c=S.b} σ_{55<S.a<65} S))