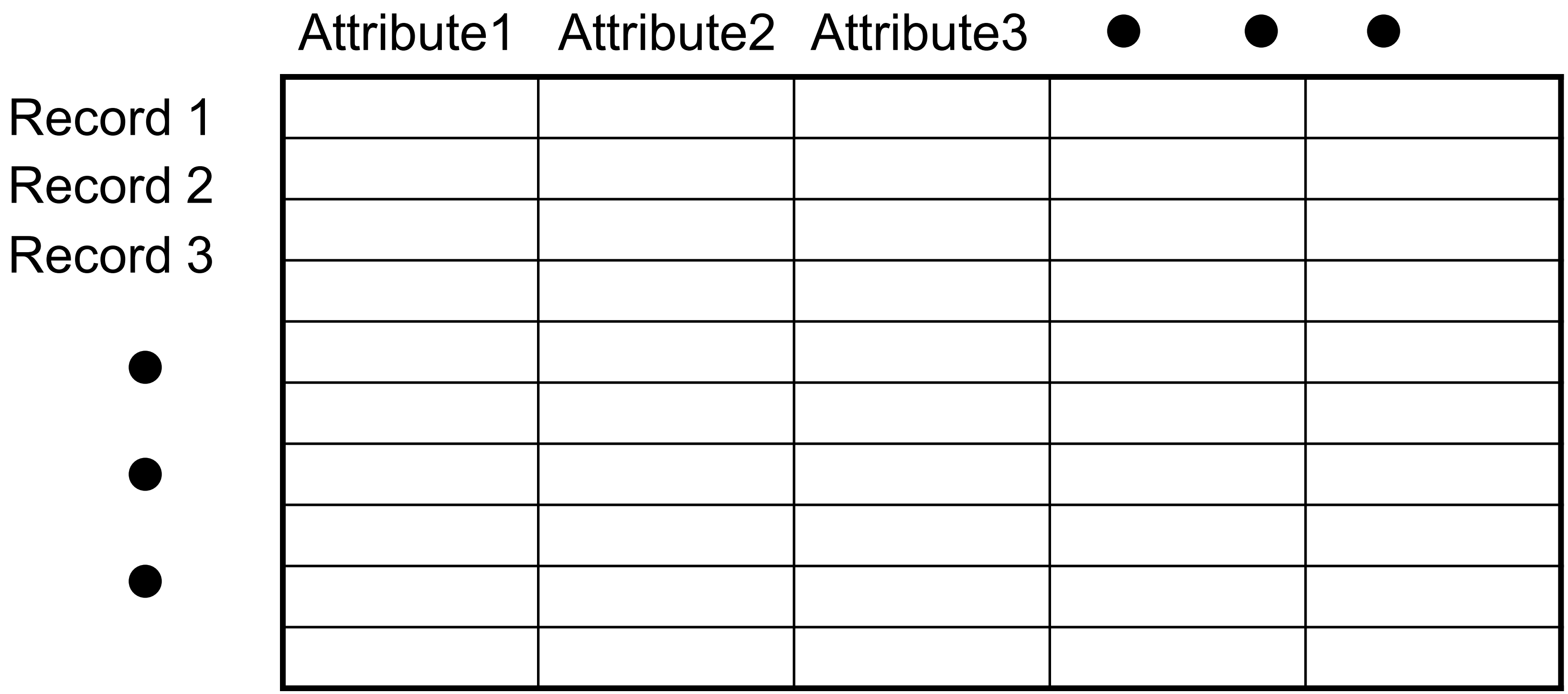
C-STORE

E0261

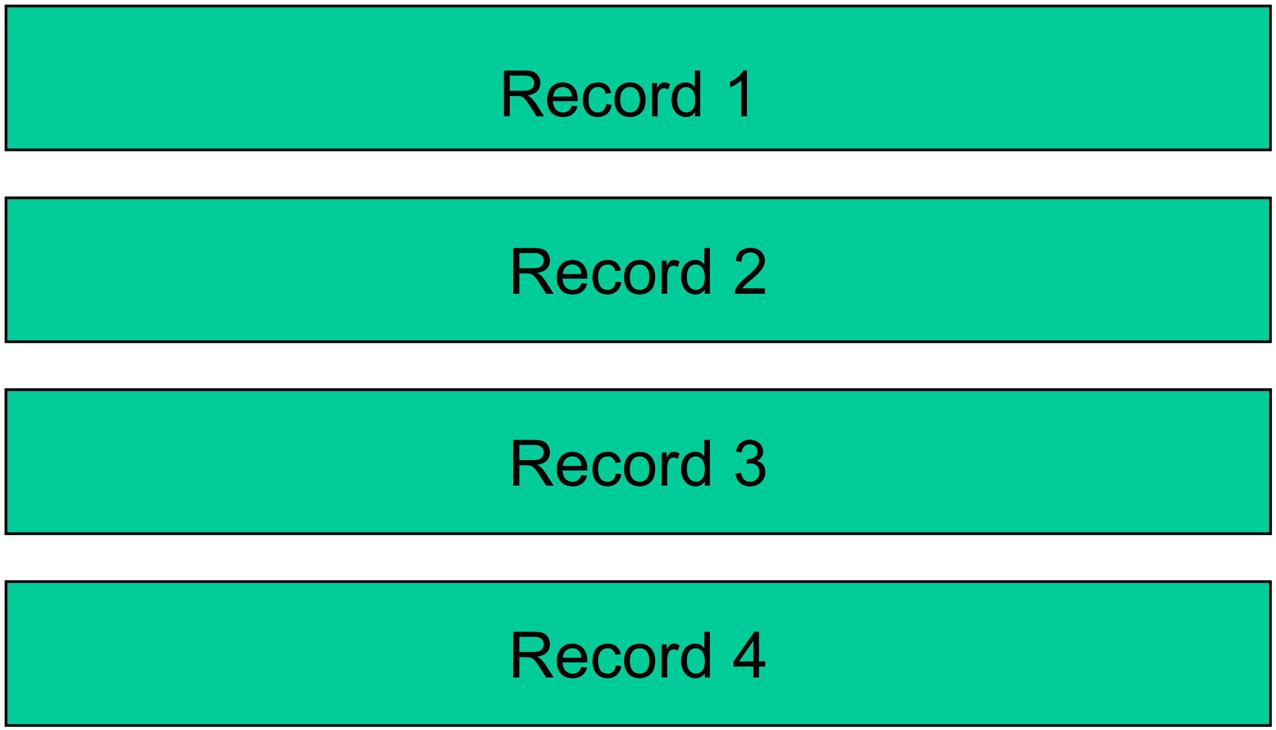
Jayant Haritsa
Computer Science and Automation
Indian Institute of Science

Relational Database



Table

Current DBMS -- "Row Store"



E.g. DB2, Oracle, Sybase, SQLServer, …

Row Stores are Write Optimized

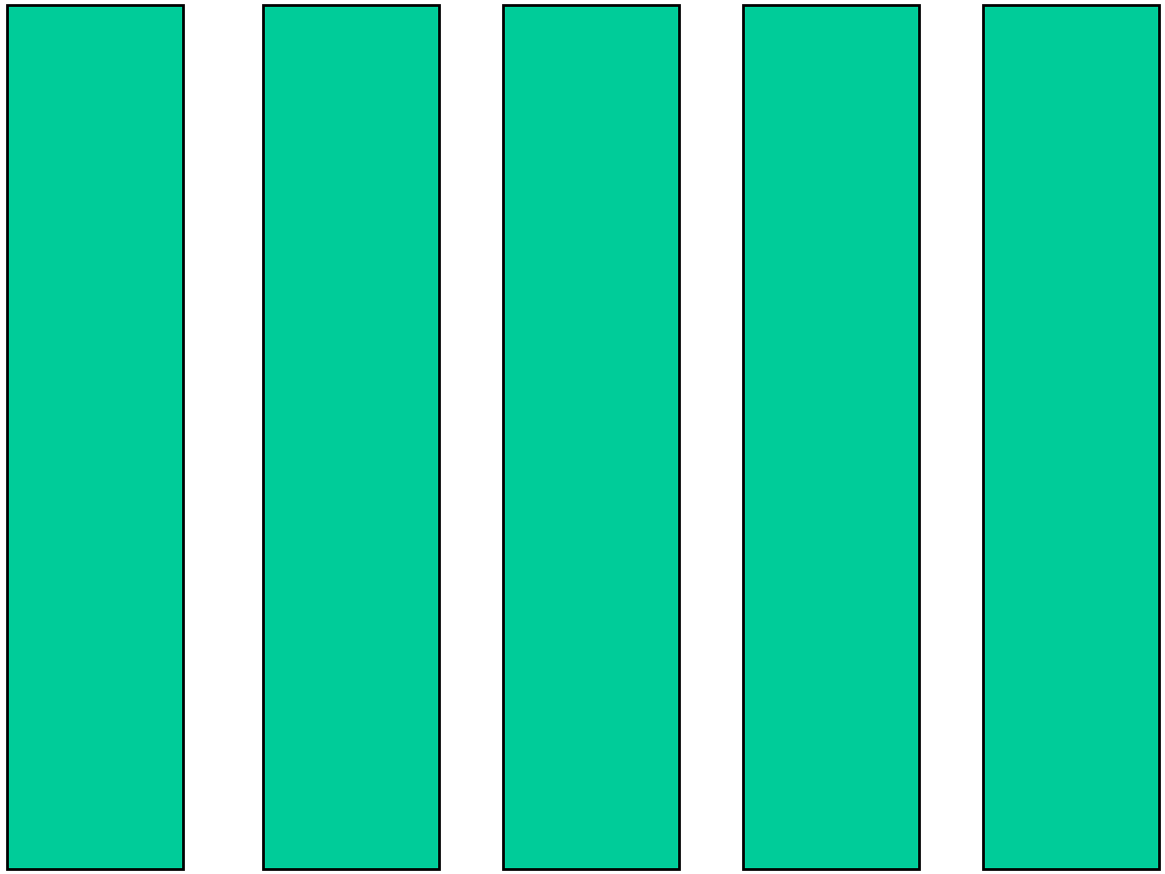
- Store fields in one record contiguously on disk
    - Can insert and delete a record in one physical write
- Use small (e.g. 4K) disk blocks
- Suitable for on-line transaction processing (OLTP) where queries typically involve
    - Small numbers of records (tuples)
    - Frequent updates
    - Many users
    - Fast response times

Does it apply to OLAP Queries as well?

- OLAP Applications: Data warehouses, Business Process Management (BPM), Electronic library catalogs, etc.
- Mostly reads but with occasional writes. eg. rare bulk loading of new data
- Few relevant columns in ad-hoc queries. On an average only 10% relevant columns in a table
- Row-store not that good for “read-mostly” applications

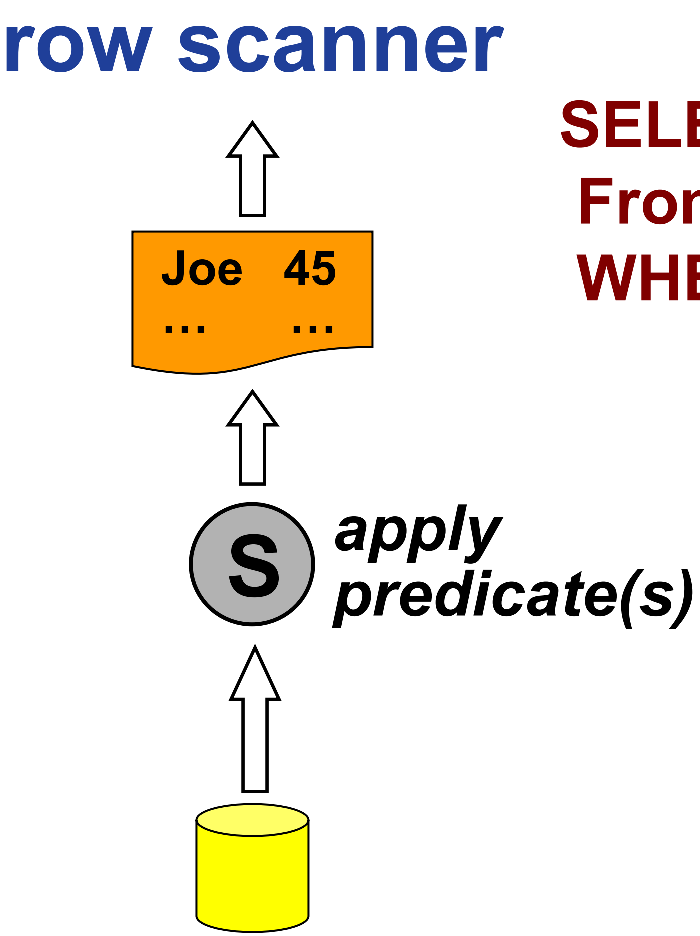
In general, better to trade off cpu cycles for disk bandwidth

What are Column Stores?

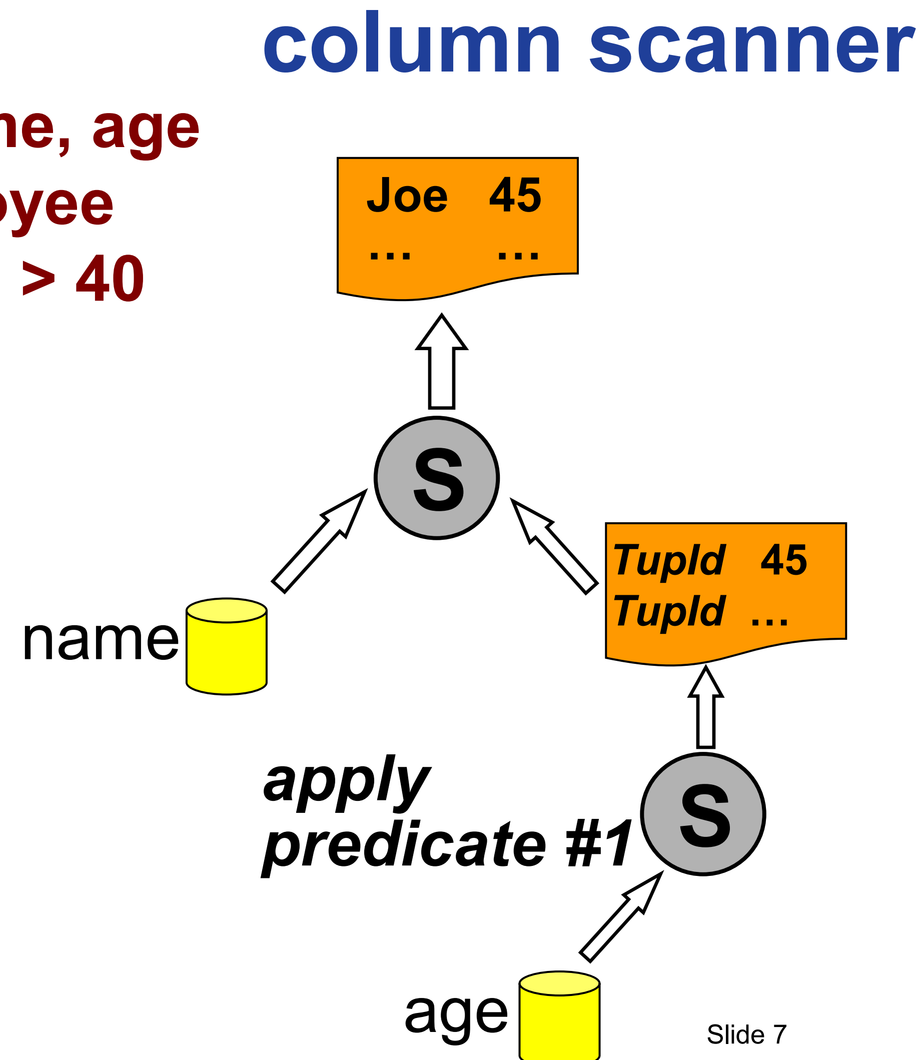


All entries of a single column are stored contiguously

Storage Engine



SELECT name, age
From Employee
WHERE age > 40



Critique of Column Stores

- Advantages
  - Fetch only required columns for a query
  - Better cache effects
  - Better compression (values within a column have same data type)
- Disadvantages
  - Incurs extra joins due to tuple reconstructions of columns from same table
  - Might need more space for storing the tuple identifiers
  - But can be slower for other applications such as
    - OLTP which could have many row inserts, …

C-Store Outline

- Data Model (Data Storage)
  - Only materialized views (perhaps many)
  - Compress the columns to save space
  - Innovative redundancy
- Handling Transactional Updates
- Query Optimization
- Performance

Data Model

- Table – collection of attributes
  - EMP(name, age, salary, dept)
  - DEPT(dname, floor)
- Projections – set of columns. Intuitively, can be thought of as materialized views
- Base tables, from which the projections are derived, are not stored physically

Contd…

- Formally, projection of a table T is a
  - A *subset* of columns of T (T could be a fact table), sorted on one (or more) columns
  - Additionally, it can contain subset of columns from other tables (such as dimension tables) which have foreign-key relationships to T
  - (conceptually) no duplicate elimination

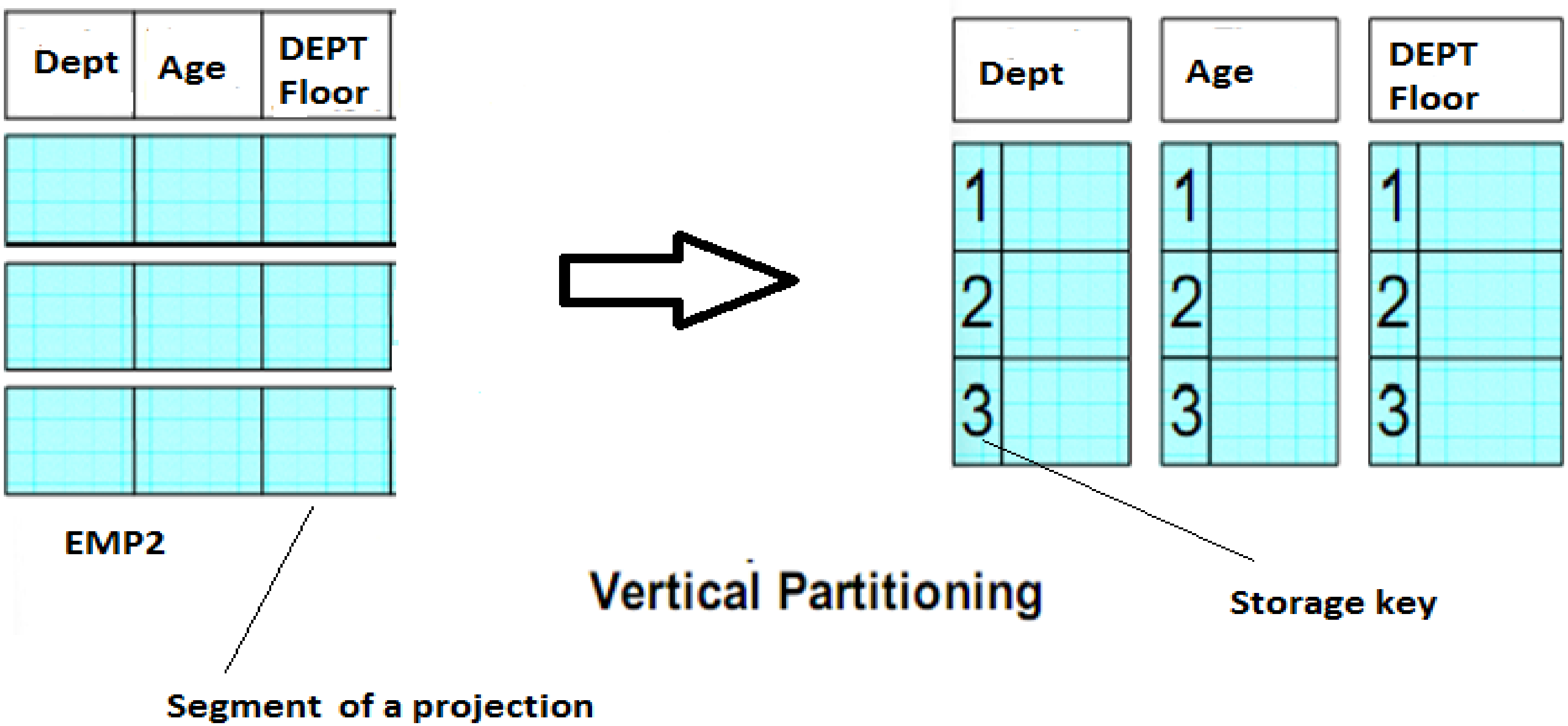
Example projections

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

Projection Details

- Each projection is horizontally partitioned into "segments"
  - Segment identifier
  - Unit of distribution and parallelism
  - Value-based partitioning, key range of sort key(s)
- Column-wise store inside each segment
- Storage Keys:
  - Within a segment, every data value of every column is associated with a unique Storage Key
  - Values from different columns with matching Storage key belong to the same logical row
  - Virtual in Read Store, Explicit in Write Store

Vertical Partitioning: Example

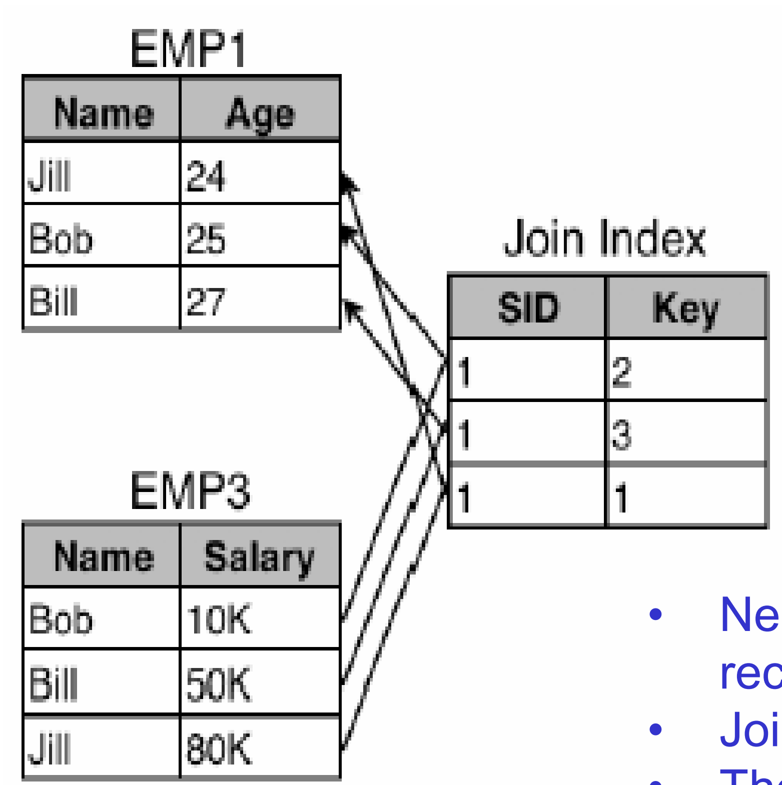


Reconstructing Base Table from Projections

- Projections are joined using Storage Keys and Join Indexes
- Join Index
  - Projection P1 has M segments, projection P2 has N segments
  - P1 and P2 are on the same base table
  - Join Index from P1 to P2 consists of
    - An entry for each row of P1 which corresponds to matching row in P2
    - Each entry is of the form (s: Segment ID in P2, k: Storage Key in the Segment s)
  - There are M join indexes for P1, one per segment
  - Value-based joins but of system identifiers ☹

Example

- Construct EMP(name, age, salary) from EMP1 and EMP3 using join index on (EMP3 to EMP1)



- Needs multiple join indices for reconstructing a base table
- Join index is costly to store and maintain
- Therefore make each column part of several projections

Compressing Columnar Data

- Different encoding schemes for different columns
- Depends on ordering and value distribution
  - (Type 1) Self-order, few distinct values
    (value, position, num_entries)   [4,12,7]
  - (Type 2) Foreign-order, few distinct values
    (value, bitmap), sparse bitmap is run-length encoded
  - (Type 3) Self-order, many distinct values
    block-oriented, delta encoding
    - 1,4,7,7,8,12 → 1, (+)3, (+)3, (+)0, (+)1, (+)4
  - (Type 4) Foreign-order, many distinct values
    leave unencoded

Redundant Storage

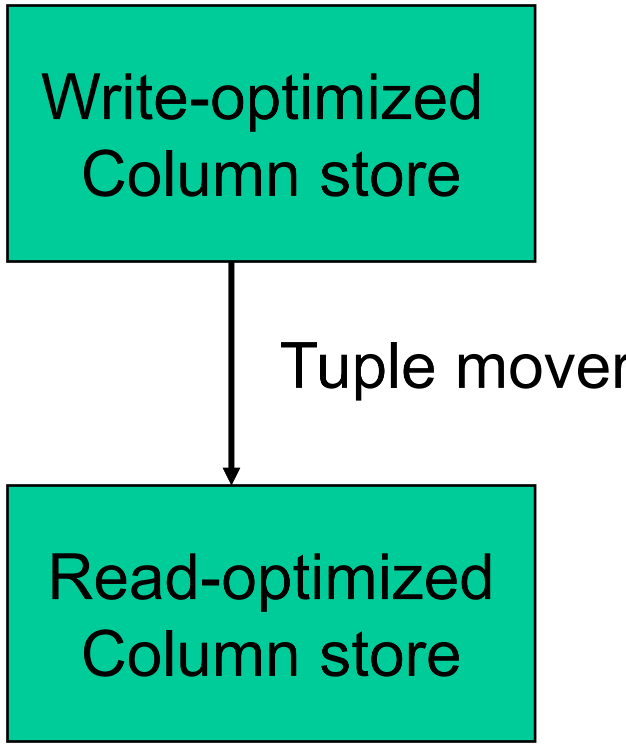
- Hardly any warehouse is recovered by redo from log
  - Takes too long!
- Store enough projections to ensure K-safety
  - Columns are in sufficient different projections and sites
- Rebuild dead objects from elsewhere in the network

Handling Transactional Updates

Updates Are Necessary

- Transactional updates are necessary even in read-mostly environment
  - Updates for error corrections
  - Real-time data warehouses
  - Rare Batch updates

Solution – a Hybrid Store



(Batch rebuilder)

(What we have been talking about so far)

Write Store

- Column store with same projections and join indexes as RS
- Horizontally partitioned as the read store
  - 1:1 mapping between RS segments and WS segments
- No compression (the data size is small)
- Each column in projection is collection of (v, sk)
- Storage keys are explicitly stored and sorted using B-tree index on sk
  - larger than the maximum in RS
- Sort-key of projection (s, sk) via B-tree indexing on s

Handling Updates

- Optimize read-only query: do not hold locks
  - Snapshot isolation
  - Query is run on a snapshot of the data as of effective time ET
    - Record visible if inserted before ET and (may be) deleted after ET
  - Ensure transactions related to snapshot are already committed
  - Coarse granularity timestamps called epochs”
- Each WS site: insertion vector (with timestamps), deletion vector, (updates become insertions and deletions)
- Maintain a high water mark and a low water mark of WS sites:
  - HWM: all transactions before HWM have committed
  - LWM: no records in RS are inserted after LWM, only deletions permitted

Handling Updates (contd)

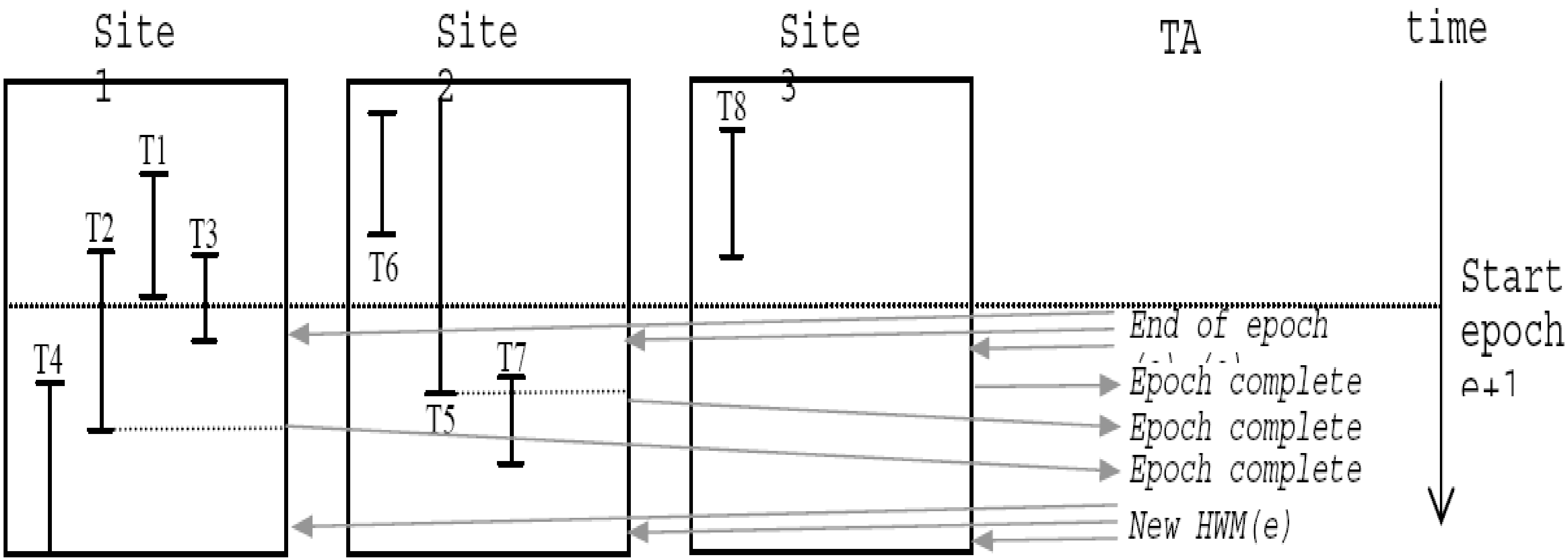
- Concurrency Control: Read-write transactions use Strict 2PL

- Recovery: Standard WAL with No-Force, Steal
  - Only log UNDO records
  - Redo from information available at other sites
  - Allows dispensing with Prepare phase in 2PC

Tuple-Mover

- Read status of the RS segment
- Record last move time for this segment from WS in T_{last_move}
- Update the RS only if insertion times $\leq$ LWM and update T_{last_move} to be most recent insertion time
- Note: All changes are done into a new version of the RS segment, i.e. updates not done in place.

HWM and Epochs



- TA: time authority updates the coarse timer (epochs)

Query Optimization

Operators

- Decompress
- Select: generates bitstring
- Mask: bitstring+projection → selected rows
- Project: choose a subset of columns
- Concat: combine multiple projections that are sorted in the same order
- Sort: All columns in a projection by some subset
- Permute: permute a projection according to a join index
- Join
- Aggregation operators and Bitstring operators

Column Optimizer

- Selinger-style cost-based query plan construction
- Chooses projections on which to run the query
- Cost model includes compression types
- Also chooses when to apply “mask” operator

Performance Results of CSTORE

Performance Benefits

- CStore is, on average, an order of magnitude faster than commercial row and column store
- Similar benefits is obtained wrt savings in storage space
- Reasons
  - Avoids reads of unused columns
  - Storing overlapping projections than the whole table
  - Better compression of data
- However, queries (Q1-Q7) are designed to suit their system! Single-column outputs, with aggregates.
- Updates not tested!

END