Presented by: Manoj Karthick Selva Kumar

C-Store: A column-oriented DBMS

MIT CSAIL, Brandeis University, UMass Boston, Brown University

Proceedings of the 31st VLDB Conference, Trondheim, Norway 2005

Authors

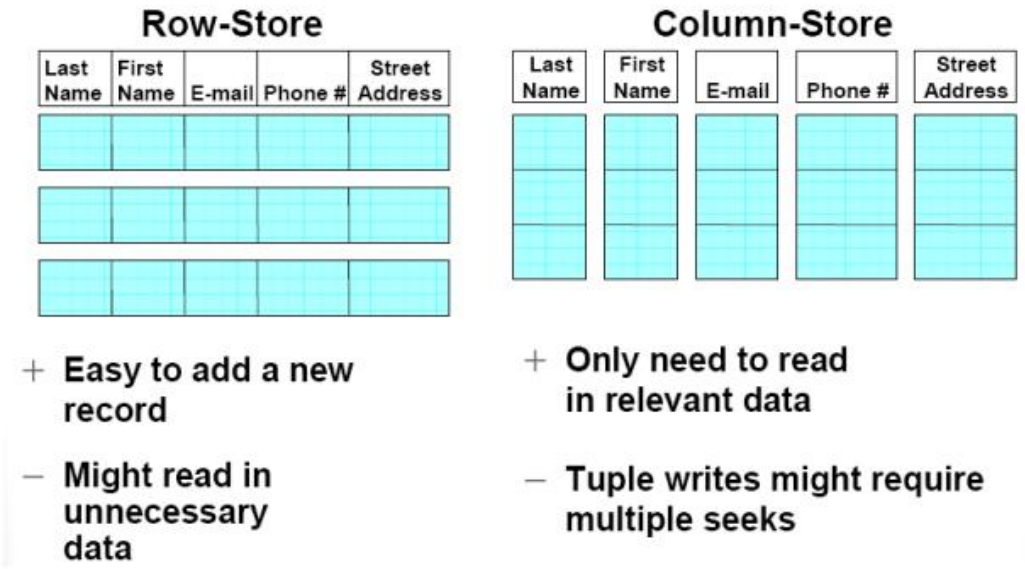
- MIT CSAIL
  - Mike Stonebraker
  - Daniel Abadi
  - Miguel Ferreira
  - Edmond Lau
  - Amerson Lin
  - Sam Madden
- Brandeis University
  - Adam Batkin
  - Mitch Cherniack
  - Nga Tran
- Umass Boston
  - Xuedong Chan
  - Elizabeth O'Neil
  - Pat O'Neil
- Brown University
  - Alex Rasin
  - Stan Zdonik

Agenda

- Why do we need Column stores?
- What is C-Store ?
- C-Store Architecture
- Important Terms and Definitions
- Data Model
- Components of C-Store
- Storage Management
- Query Execution and Optimization
- Performance Benchmarking

Why do we need Column Stores?

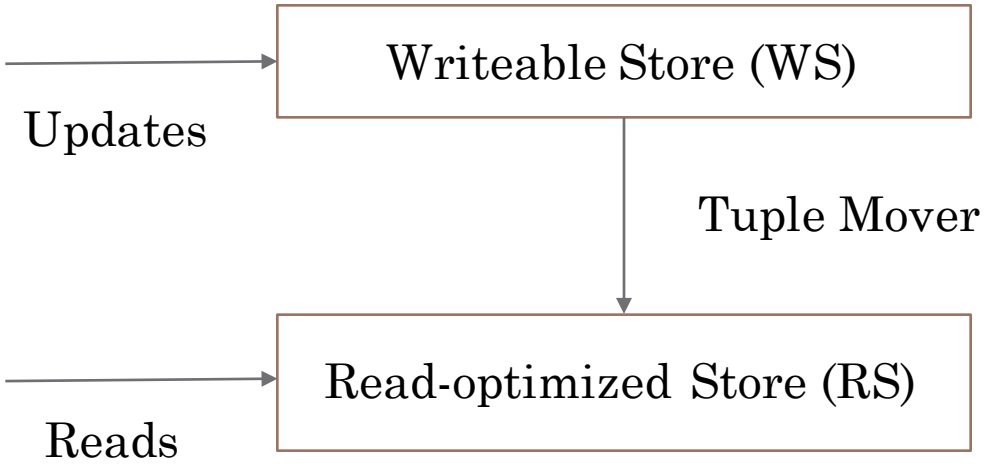
- In current Row Store architectures:
  - Contiguously store attributes of a record (tuple)
  - One write
  - High performance writes
  - Use cases: OLTP style applications
- Need Ad-hoc computation of aggregates
- Pre-computation not feasible/possible
- Need → Read-Optimized systems, in which:
  - Single column stored contiguously
  - Read only the required columns
  - Densepacked storage
  - Trade-off between Disk bandwidth and CPU Cycles
- Use cases: CRMs, Library Catalogs, Ad-hoc inquiry systems



What is C-Store ?

- Column-oriented data store
- Stores collection of columns sorted on attributes
- High retrieval performance
- High Availability
- Works on off-the-shelf hardware

Architecture of C-Store



Architecture Reasoning

- Hybrid architecture containing WS and RS
- Need for hybrid architecture
  - Historically need to compromise between high performance online updates vs read-only optimizations
  - Example: Warehouses need on-line updates, CRMs need real-time data visibility
- C-Store approach:
  - Include Writeable store for high-performance updates, inserts and deletes
  - Read-optimized store for high-performance read only queries
  - Usage of tuple mover to move records from WS to RS.
- Provide best of both worlds
- Focus on read performance than writes
- **Result**: Support large ad-hoc queries (RS), smaller update transactions (WS) and continuous inserts (Tuple mover).

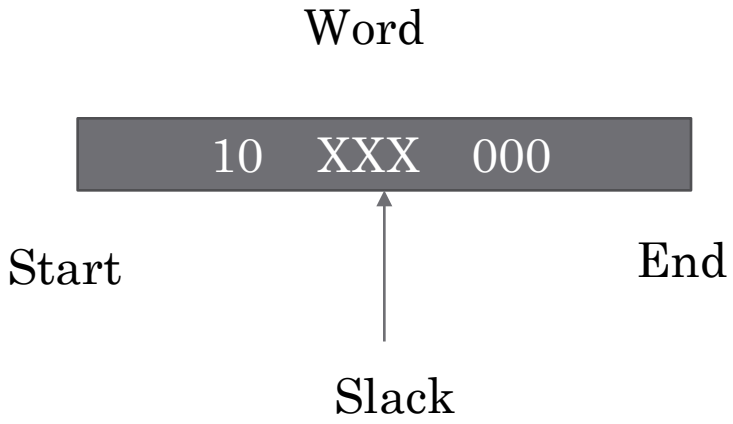
Innovative features of C-Store

- Hybrid architecture with read and write optimized components
- Redundancy through storage of overlapping projections
- Heavily compressed columns
- Column-oriented optimizer and execution
- High availability
- Fault tolerance
- Snapshot Isolation

Important Terms

- Word boundaries
- Densepack
- Projections
- Logical tables
- Anchoring
- Sort keys
- Horizontal Partitioning
- Segment Identifiers
- Storage Keys
- Key ranges
- Join Indexes
- K-Safety
- Paths
- Space budget

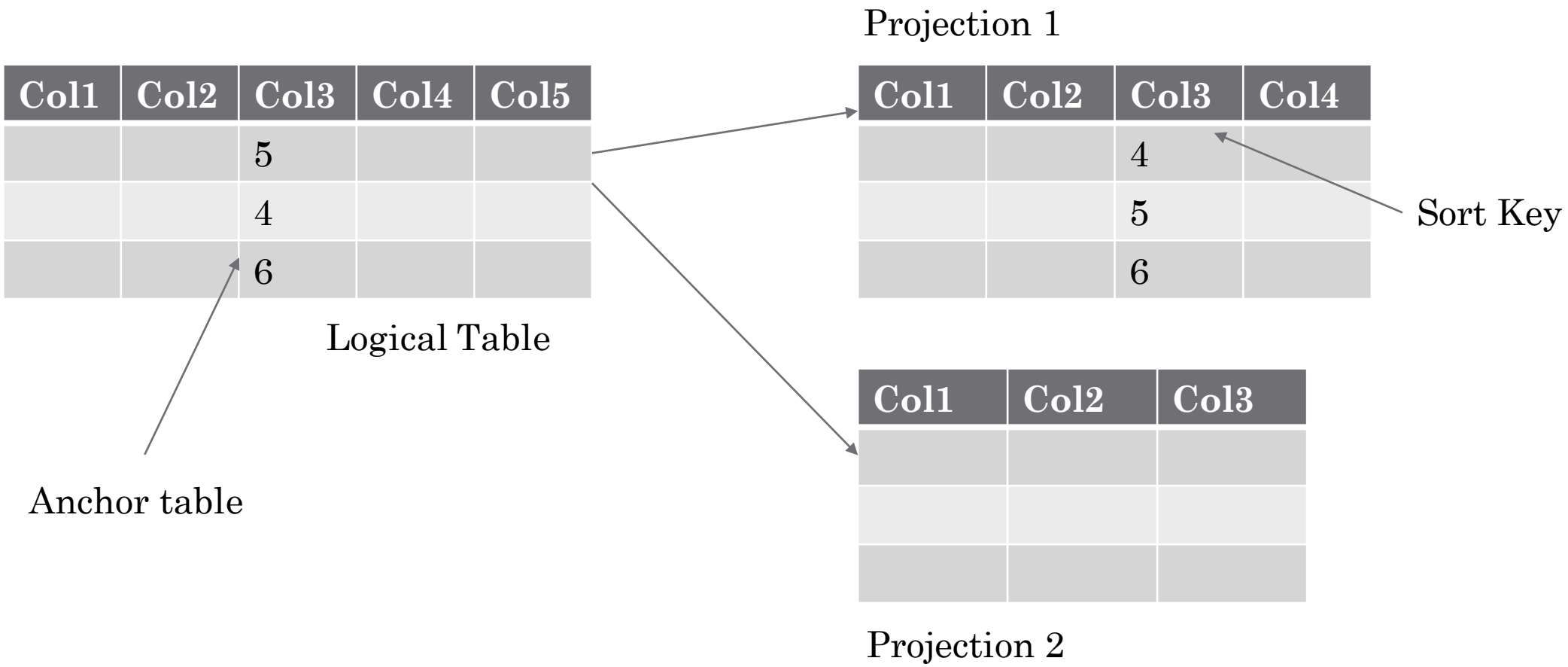
Word boundaries, Densepack



C-Store Data model

- Supports relational logical data model
  - Consisting of named collection of attributes (columns)
  - Primary keys
  - Secondary keys
  - Foreign keys
- Implements only projections
- Projections are anchored on a table T
- Base table T is not stored
- How to form a projection?
  - Project attributes
  - Retain duplicate values
  - Perform joins
  - Sort

Projections, Logical Tables, Anchoring, Sort Keys



C-Store data model

Name	Age	Dept	Salary
Bob	25	Math	10K
Bill	27	EECS	50K
Jill	24	Biology	80K

Sample employee table (EMP)

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

Possible Projections

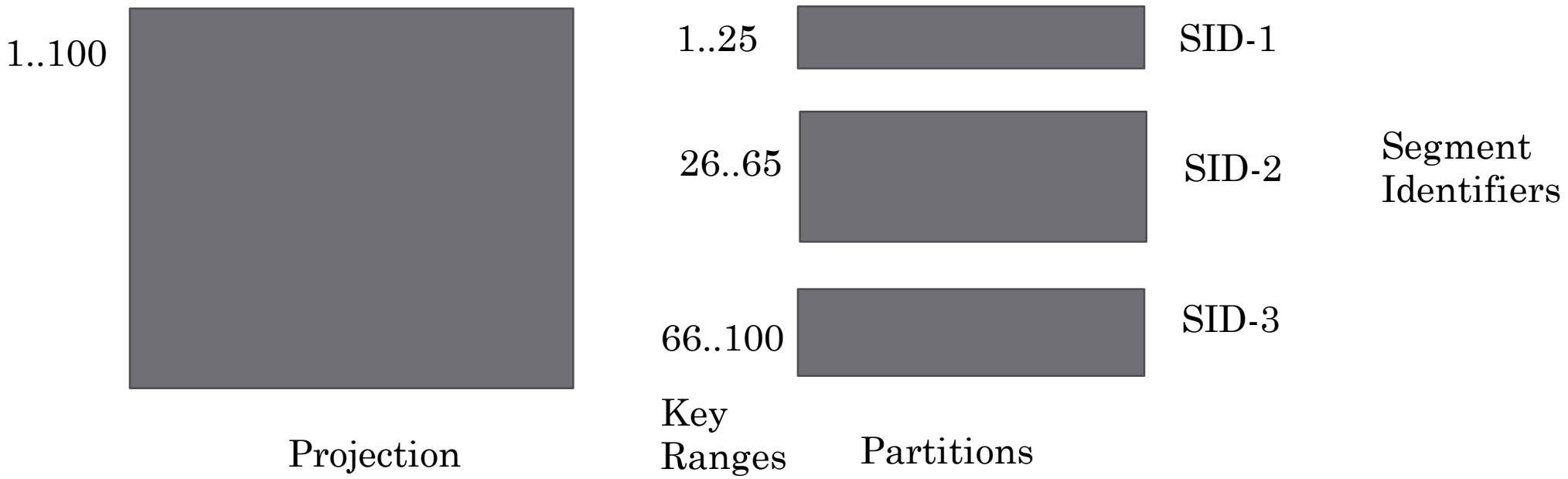
EMP1 (name, age | age)
EMP2 (dept, age, DEPT.floor | DEPT.floor)
EMP3 (name, salary | salary)
DEPT1 (dname, floor | floor)

Possible Projections with Sort keys

C-Store Data model

- Every projection:
  - Horizontally partitioned into segments
  - Support only value based projections
  - Has an associated key range
- Need for covering set of projections
- Supports grid based systems
- Reconstruction of complete rows of tables needed. Done through:
  - Storage Keys
  - Join Indices

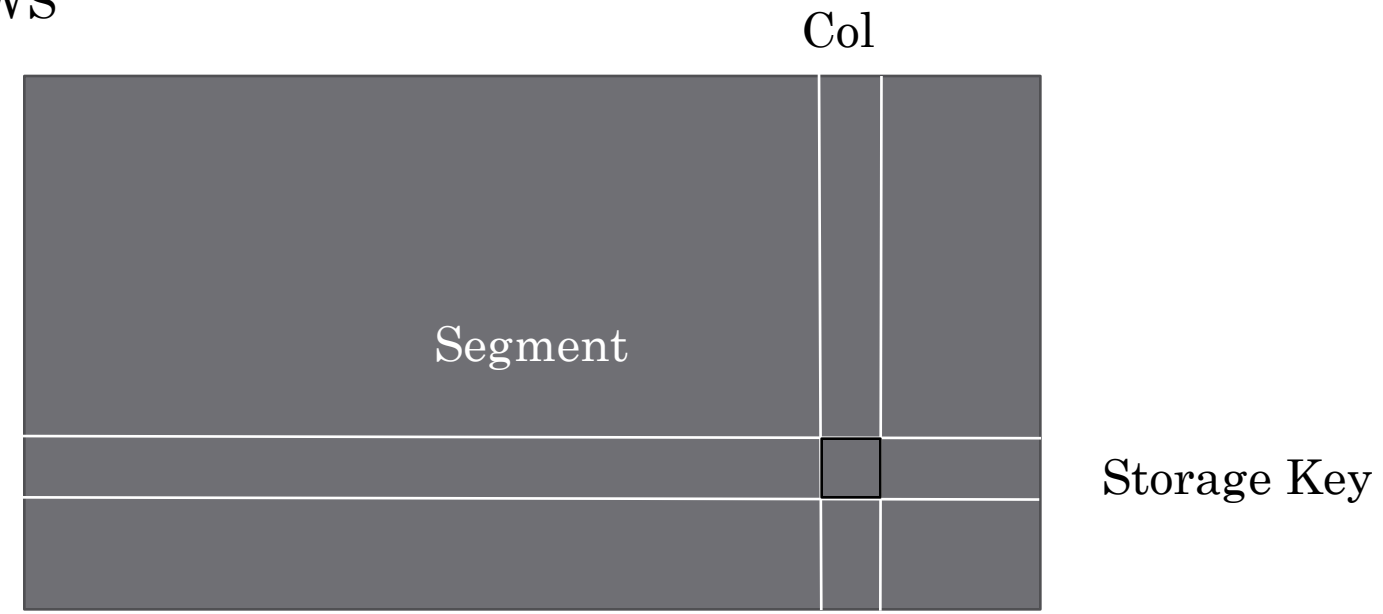
Horizontal Partitioning, Segment Identifiers, Key range



Storage Key

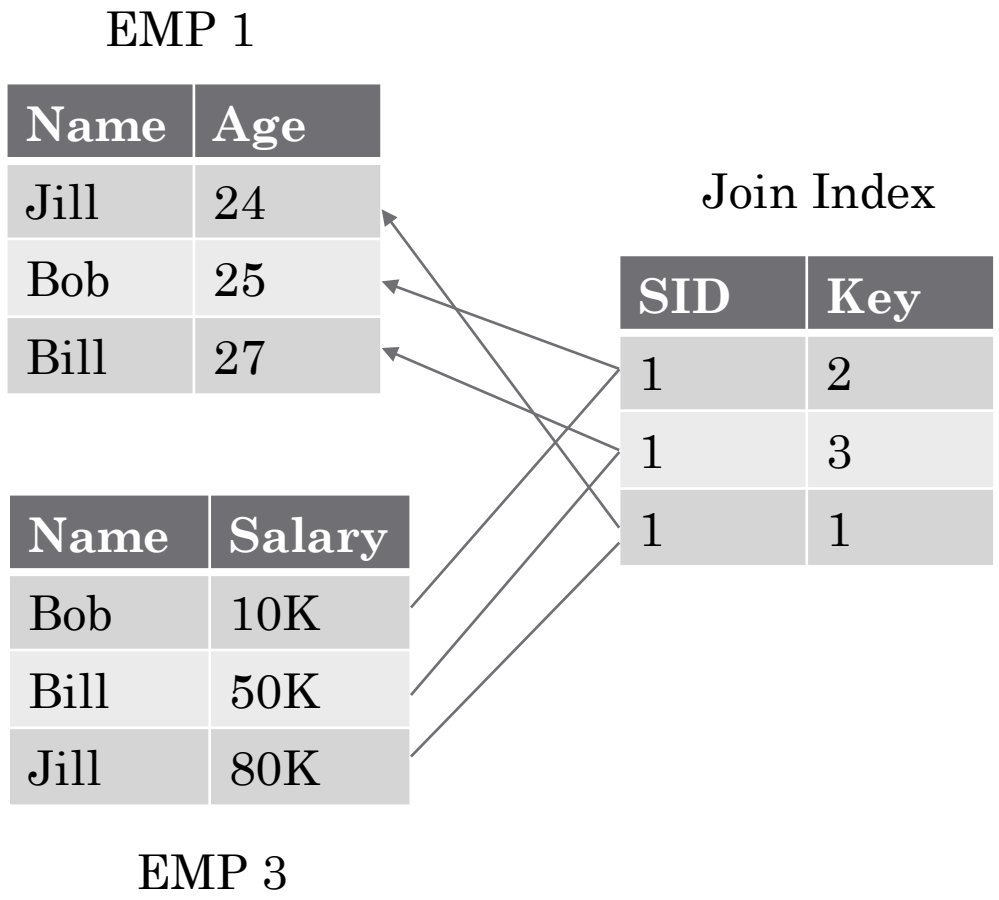
- Every data value in every column → Storage Key
- Logically identifies Row
- Inferred in RS; Stored in WS

Different Columns + same SK = Same row



Join indices

- Reconstruction of row/tuple
  - Join segments from different projections
  - Using Join Indices and Storage Keys
- Definition:
  - If T1 and T2 are two covering set of projections of table T, a join index from M segments of T1 to N segments of T2 is logically a collection of M tables, one per segment S, of T1 consisting of rows of the form *(s: SID in T2, k: Storage key in S)*
- Join indices are always 1:1



Join indices

- Expensive to store
- In reality, expect to store multiple overlapping projections
- Thus, only few join indices are required
- Every modification of projection = Update of Join Index

- To reconstruct logical table T from set of projections, need to define paths.
- Path
  - A collection of join indices originating with some sort order Ti, passes through zero or more intermediate join indices and ends with projection stored in some order O*

K-safety

- Allocation of Segments of projections and corresponding join indexes into various nodes
- Admin can specify tables be K-safe
- K-safe = the database can suffer max ‘k’ failures
- Despite of failed sites, covering set needs to exist.
- In other words, # of bearable failed sites = K

How to determine projections, segments, sort keys and covering set?

- This collection needs to support K-safety
- Usage of training workload – to be provided by C-Store admin
- Space Budget constraints
- Automatic schema design tool being developed

RS

- Read optimized column store.
  - Segments consists/ broken into columns.
  - Column stored in order of sort key
  - Trade-off between entry based sorting vs sort-key based sorting
- Encoding Schemes:
  - Type 1: Self Order, Few distinct values
  - Type 2: Foreign Order, Few distinct values
  - Type 3: Self Order, Many distinct values
  - Type 4: Foreign Order, Many distinct values
- Explore each of these in upcoming slides

Type 1: Self Order, Few distinct values

- Columns represented as (v, f, n) where:
  - V = value stored in column
  - F = position in column where v first appears
  - N = number of Vs in the column

..... 4 4 4 4 4 4 4 .....

Pos 12-18

- One triple for every distinct value / column
- Usage of B-tree indices over columns' value fields for search indexing
- Example:
  - Column A, group of 4's in positions 12-18
  - Encoded as (4, 12, 7)

Type 2: Foreign Order, Few distinct values

- Columns represented as (v, b) where:
  - V = value stored in the column
  - B = bitmap indicating the positions in which v is stored
- For search, usage of B-trees that map positions to values

Bitmap Indexes

Column C	V = 0	V = 1	V = 2
0	1	0	0
1	0	1	0
2	0	0	1
0	1	0	0

Type 3 and Type 4

- Type 3: Self Order, Many distinct values
  - Column represented as Delta from the previous value in the column
- Type 4: Foreign Order, Many distinct values
  - No efficient encoding available as of 2005
- Decompression happens in main memory

Col C	1	4	7	7	8	12
Col (Del-C)	1	3	3	0	1	4

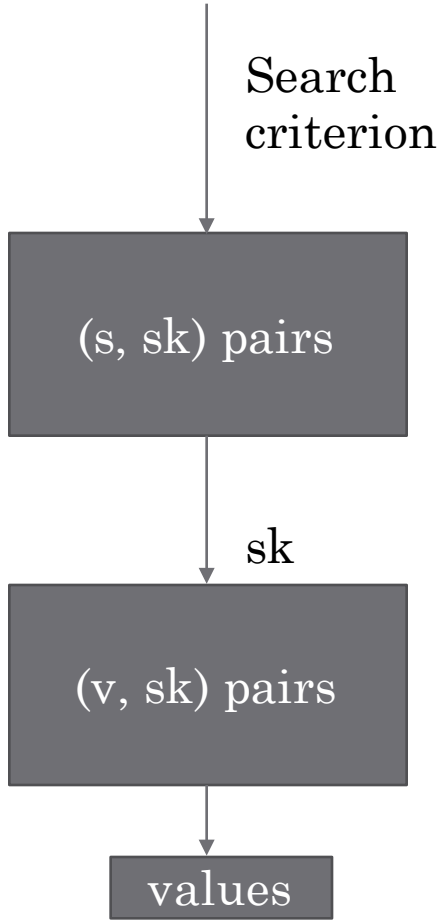
Type 3 encoding

WS

- Physical design same as RS
- Projections and Join Indices are similar as in RS
- WS is horizontally partitioned in same way as RS; results in 1:1 mapping between RS and WS segments
- Storage is different and revamped for efficient updates to the database
  - SK is stored explicitly instead of inference
  - Unique SK given to each logical insert
- Storage Representation:
  - Uncompressed
  - Represented as collection of (v, sk) pairs
  - v = value in the column
  - sk = corresponding storage key
  - Represented as conventional B-tree based on sk's value

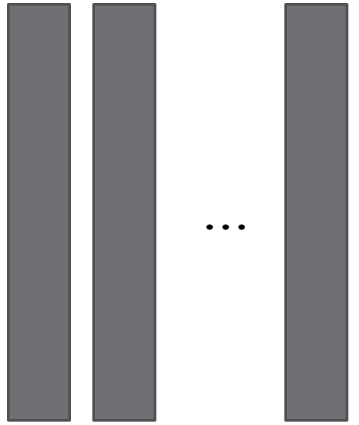
WS

- Sort keys of each projection in WS
- Represented as a collection of (s, sk) pairs
  - s: a particular sort key value
  - sk: Storage key corresponding to where the sort key value first appears
  - Represented as conventional B-tree based on sort key field
- Search in WS:
  - Step 1: Use B-tree over the (s, sk) pairs
    - Search condition: search criteria of the query
    - Search result: corresponding storage keys
  - Step 2: Use B-tree over the (v, sk) pairs
    - Search condition: storage key sk found in previous step
    - Search result: data values



Storage Management

- Need to allocate segments to nodes in grid system
- How ? A “Storage allocator”
- Design constraints:
  - All columns of single segment should be co-located
  - Join indexed co-located with “sender” segments
  - WS segments co-located with RS segments for same key range



Columns of segment

Co-location Trifecta

Storage Management

- Need to design storage allocator with mentioned constraints in mind
- Storage allocator needs to:
  - Perform initial allocation
  - Re-allocation when data in unbalanced
- Being worked on as of 2005; details of allocator beyond scope of the paper

Updates and Transactions

- In C-Store, updates are represented as an insert followed by a delete.
  - Each object is inserted in all projections containing the given column
  - If the inserts correspond to the same logical record, they would have the same storage key
- Synchronization of Storage keys
  - To avoid synchronization between the all the grid sites, every site maintains a locally unique counter; append Site ID to the counter making it globally unique
  - Storage keys in WS consistent with RS
    - Set initial counter of WS = Largest Key in RS + 1
- WS built on top of BerkeleyDB
  - leverages data structure provided by the database

Snapshot Isolation

- Isolate read-only transactions by allowing reads in at some point in the recent past
- Need to find:
  - High water mark: Time before which we can guarantee there are no uncommitted transactions
  - Low water mark: Earliest effective time at which a read-only transaction can run
- To provide snapshot isolation, there should be no in-place updates
  - A record is visible if:
    - Inserted before the LWM
    - Deleted after the HWM
- To calculate effective time without keeping track of time at every site at the atomic level, use coarse granulated “epochs”

IV and DRV

- IV refers to the insertion vector
- DRV refers to the deleted record vector
- For every projection record for every projection, we store the IV and DRV
  - We can infer if a record is part of a snapshot using the IV and DRV
- Both IV and DRV values are stored as either 1 or 0 for a particular epoch
- Since the IV and DRV are sparse, they can be compacted using Type 2 encoding as discussed earlier.

Concurrency Control

- Uses strict 2 phase locking techniques.
- Each site sets locks on data objects that are read or written to by the system.
- This forms a distributed Lock table.
- Distributed COMMIT processing:
  - A variation of 2 phase locking is used.
  - Each transaction has a MASTER that assigns work to appropriate sites and assigning commit status
  - No PREPARE messages are sent
  - When master receives COMMIT message for a particular transaction:
    - Waits until all the site workers have completed outstanding transactions
    - Then issues COMMIT or ABORT message to each site
    - Locks associated with the transaction can then be released

Failure and Recovery

- Three types of site failure:
  - No data loss – Roll forward from previous state through data from other sites
  - Both RS and WS destroyed – Reconstruction from other projections and join indices available at remote sites; Need functionality to retrieve DRV and IV to match state.
  - WS is damaged, RS intact – Out of scope
  - RS usually expected to escape damage since written to only by the tuple mover

Tuple Mover

- Moves blocks of tuples from WS to RS (Merge out process)
- Uses LSM trees to efficiently support high speed ordered tuple movement
- Also responsible for updating the Join indices
- Two types of records with Insertion time at or before LWM:
  - Ones deleted at or before LWM: Discarded by MOP
  - Ones that are not deleted or deleted post LWM: Moved to RS
- MOP creates new RS segment RS'. How ?
  - Reads blocks from columns in RS
  - Deletes any RS items with DRV set
  - Merges data from WS with RS creating RS'
  - This also means the storage keys in RS' different from RS → Join index maintenance reqd.

Queries

- Decompress
- Select – returns bitstring repr.
- Mask – emits where 1
- Project
- Sort
- Aggregation Operators
- Concat
- Permute
- Join
- Bitstring Operators – boolean AND/OR
- **NOTE:** Many queries are optimized such that they can operate on the compressed version rather than needing to decompress

Performance Comparison

- Performance benchmarked on TPC-H data
- Read only queries, only on RS

```
CREATE TABLE LINEITEM (
L_ORDERKEY  INTEGER NOT NULL,
L_PARTKEY   INTEGER NOT NULL,
L_SUPPKEY   INTEGER NOT NULL,
L_LINENUMBER       INTEGER NOT NULL,
L_QUANTITY  INTEGER NOT NULL,
L_EXTENDEDPRICE    INTEGER NOT NULL,
L_RETURNFLAG       CHAR(1) NOT NULL,
L_SHIPDATE  INTEGER NOT NULL);

CREATE TABLE ORDERS  (
O_ORDERKEY  INTEGER NOT NULL,
O_CUSTKEY   INTEGER NOT NULL,
O_ORDERDATE INTEGER NOT NULL);

CREATE TABLE CUSTOMER (
C_CUSTKEY   INTEGER NOT NULL,
C_NATIONKEY INTEGER NOT NULL);
```

Result

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

Q1. *Determine the total number of lineitems shipped for each day after day D.*

```
SELECT l_shipdate, COUNT (*)
FROM lineitem
WHERE l_shipdate > D
GROUP BY l_shipdate
```

Q2. *Determine the total number of lineitems shipped for each supplier on day D.*

```
SELECT l_suppkey, COUNT (*)
FROM lineitem
WHERE l_shipdate = D
GROUP BY l_suppkey
```

Q3. *Determine the total number of lineitems shipped for each supplier after day D.*

```
SELECT l_suppkey, COUNT (*)
FROM lineitem
WHERE l_shipdate > D
GROUP BY l_suppkey
```

Q4. *For every day after D, determine the latest shipdate of all items ordered on that day.*

```
SELECT o_orderdate, MAX (l_shipdate)
FROM lineitem, orders
WHERE l_orderkey = o_orderkey AND
      o_orderdate > D
GROUP BY o_orderdate
```

Q5. *For each supplier, determine the latest shipdate of an item from an order that was made on some date, D.*

```
SELECT l_suppkey, MAX (l_shipdate)
FROM lineitem, orders
WHERE l_orderkey = o_orderkey AND
      o_orderdate = D
GROUP BY l_suppkey
```

Q6. *For each supplier, determine the latest shipdate of an item from an order made after some date, D.*

```
SELECT l_suppkey, MAX (l_shipdate)
FROM lineitem, orders
WHERE l_orderkey = o_orderkey AND
      o_orderdate > D
GROUP BY l_suppkey
```

Q7. *Return a list of identifiers for all nations represented by customers along with their total lost revenue for the parts they have returned.* This is a simplified version of query 10 (Q10) of TPC-H.

```
SELECT c_nationkey, sum(l_extendedprice)
FROM lineitem, orders, customers
WHERE l_orderkey=o_orderkey AND
      o_custkey=c_custkey AND
      l_returnflag='R'
GROUP BY c_nationkey
```

Performance Comparison

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

Using Non-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

Using Materialized Views


Questions?