C-Store: A Column-oriented DBMS

Authors: Mike Stonebraker, Daniel J. Adabi, Adam Batkin, Xuedong Chen, etc.

Presenter: Qingyuan Feng

This paper has won 2015 VLDB 10-Year Best Paper Award

Agenda

- Why column store?
- Intro to C-Store
- Data Model
- RS (Read-optimized column Store)
- WS (Writeable Store)
- Storage Management
- Updates and Transactions
- Tuple Mover
- Query Execution
- Performance Comparison
- Conclusions

What is Column Store? Why Column Store?

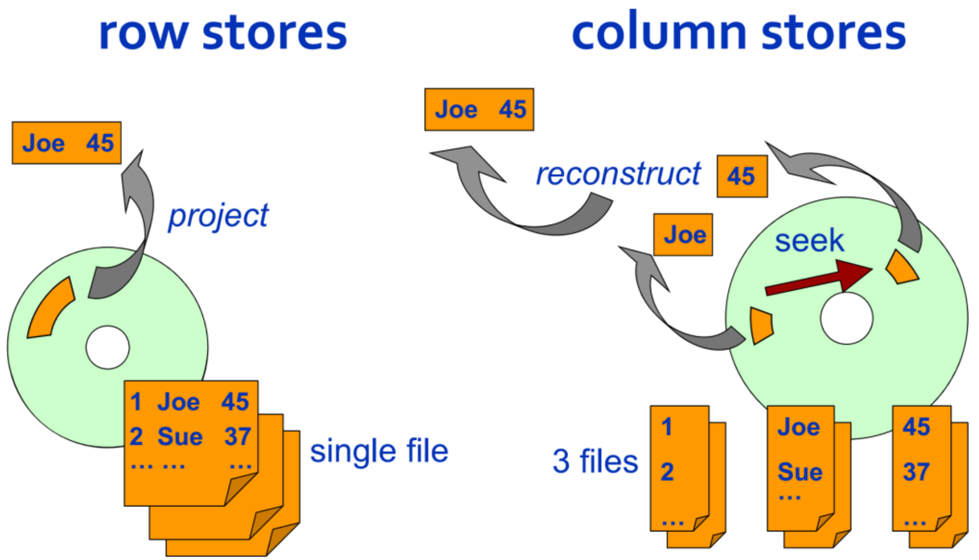
- Most major DBMS are record-oriented, called row store.
  - Attributes of a tuple are placed contiguously in storage.
- Row store systems are called write-optimized.
  - Effective for OLTP-style apps.
- Querying large amounts of data requires read-optimization
  - Data Warehouses, CRM, etc.

Name	Age	City of Res.
George	35	Toronto
Barack	48	Miami
Donald	72	New York

http://www.123rf.com/photo_18101642_3d-render-of-man-placing-crm--customer-relationship-management--cubes.html

What is Column Store? Why Column Store?

- Column Store: values of single columns are stored contiguously.
  - Efficient for read-mostly apps.
- Already products existing, like Sybase IQ, Addamark, KDB.
- Avoid reading irrelevant attributes, as of row store. Good Performance.



http://www.eecs.berkeley.edu/~kubitron/cs262

Why Column Store?

- Column store trades abundant CPU cycles for disk bandwidth
  - Row store pads attributes, and store values in native format
  - Column store encode data more compactly
  - Column store densepacks values in storage.
- Row store DBMSs store complete tuples along with auxiliary B-tree indexes
  - While bit map indexes or cross table indexes are better read-optimized.
- C-Store is proposed.

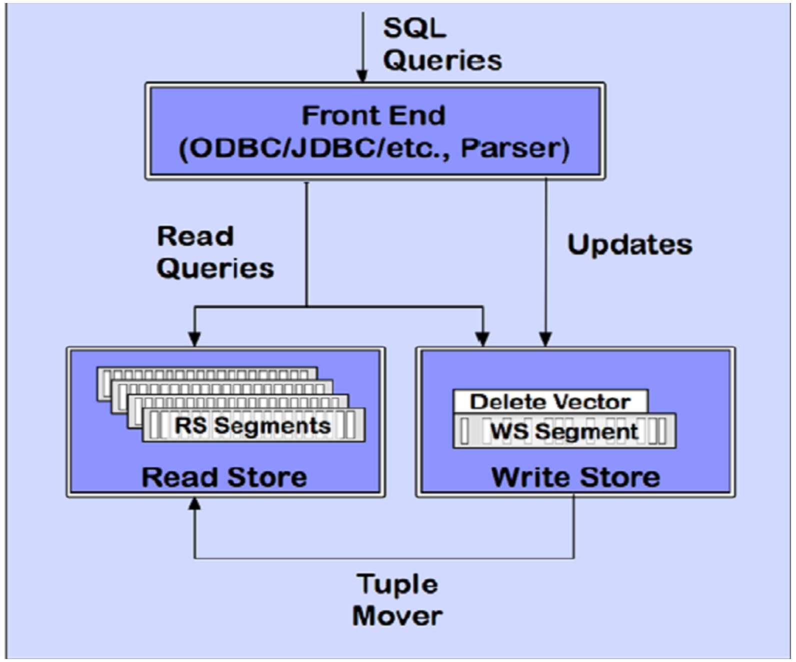
Introduction to C-Store

Introduction to C-Store

- Stores a collection of columns, each sorted on some attributes.
  - Groups of columns sorted on the same attribute are called “**projections**”
  - Same columns exist in multiple projections, sorted on different attributes.
- Designed for grid computers
  - Horizontal partitioning, intra-query parallelism.
- Storing overlapping projections improves reliability in site failures
- Problem in existing DBMS: storage sequence cannot both benefit reading and inserting.
- Solution: a novel system design.

Introduction to C-Store

- A novel system design:
  - Writeable Store (WS), Read-optimized Store (RS), Tuple Mover (TM)
  - Inserts are sent to WS, deletes are marked in RS for purging by TM.
- Snapshot Isolation based on timestamps ensures consistency
- Column-oriented optimizer and executor



http://www.eecs.berkeley.edu/~kubitron/cs262

Introduction to C-Store: innovative features

- Hybrid architecture (WS and RS)
- Redundant storage in overlapping projections of different orders
- Compressed columns using coding
- Column-oriented optimizer and executor
- High availability achieved through overlapping projections
- Snapshot isolation to avoid 2PC and locking

Data Model

Data Model

- Standard relational logical data model
  - Attributes can form primary key or foreign key
  - Implements **only projections**, which is **anchor**ed on a given logical table and can have >=1 attributes, can overlap with other tables
- Example: EMP(name, age, salary, dept) and DEPT(dname, floor) relations
- K attributes in a projection, K structures, each data structure sorted on the same sort key

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

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

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

Data Model

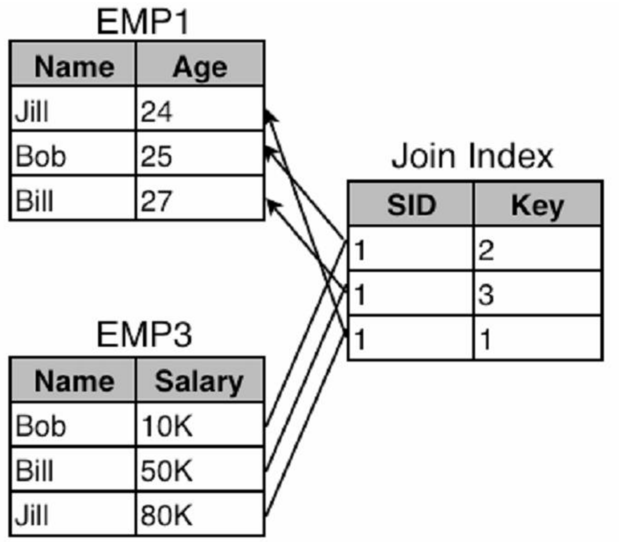
- Projections are horizontally partitioned into segments, and given segment ID (sid).
  - Each segment is associated with a key range of sort key
  - Must join segments from different projections to reconstruct tables
- **Storage Keys** (SK)
  - Each segment, every entry of every column has an SK
  - Can form logical rows
- **Join Indices**
  - Example: join index from M segments in T1 to N segments in T2
  - Logically a collection of M tables.

`(s: SID in` *`T2`*`, k: Storage Key in Segment s)`

Data Model

- To construct table T from T1, …, Tk, find a path through join indices
  - Example: reconstruct EMP table from the projections
  - EMP2 => EMP1; EMP3 => EMP1
  - EMP2 => EMP3; EMP3 => EMP1
- Example: EMP3 => EMP1
  - Assume the records in EMP1 are in Segment 1.

(s: SID in *T2*, k: Storage Key in Segment s)




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

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

Data Model

- In practice, each column is stored in several projections.
  - Fewer join indices
  - Less computational cost
- Segments of projections and join indexes are allocated to nodes
  - Notion of K-safe
- Design problem: representation of projections, segments, sort keys and join indices
  - K-safety
  - Can keep log

RS (Read-optimized Store)

RS (Read-optimized Store)

- Each column stored in order of sort key
- Storage key: ordinal number, calculated
- Encoding Schemes

Order	No. distinct values	Type
Self	few	1
Foreign	few	2
Self	many	3
Foreign	many	4

RS: Encoding Schemes

- Type 1: self-order, few distinct values
  - A sequence of triples, (v, f, n)
  - Example: a group of 4's appears in positions 12-18, represented by (4, 12, 7)
  - B-tree indexes over **value** fields for search queries.
  - Can densepack the index, leaving no empty space.
- Type 2: foreign-order, few distinct values
  - Represented by a sequence of tuples, (v, b), b is bitmap indicating positions.
  - Example: a column of integers (0, 0, 1, 1, 2, 1, 0, 2, 1), can be represented as 3 pairs: (0, 110000100), (1, 001101001), (2, 000010010).

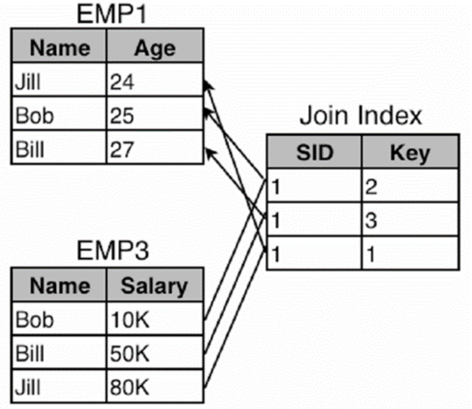
RS: Encoding Schemes

- Type 3: self-order, many distinct values
  - Represent every value in the column as a delta from the previous value in the column
  - Example: a column of values 1, 4, 7, 7, 8, 12 is represented by 1, 3, 3, 0, 1, 4
  - First entry of every block is a value in the column and its SK, subsequent values are deltas from the previous one.
  - Can densepack the index, B-tree.
- Type 4: foreign-order, many distinct values
  - Currently left values unencoded.
  - Densepack B-tree.

WS (Writeable Store)

WS (Writeable Store)

- Identical DBMS design as RS
- Storage representation very different
  - Storage key (SK) explicitly stored in each segment
  - 1:1 mapping between RS segments and WS segments
  - Data represented without compression, use B-tree indexing
  - Columns: sequences of (v, sk), v: value, sk: storage key. Sort keys are additionally represented by (s, sk), where s is sort key.
- Join indices
  - Each join index is stored with the associated "sending" record.



Storage Management

- Storage allocator: segments => nodes in grid
  - All columns in a single segment of a projection are co-located.
  - Join indexes co-located with sender.
  - WS co-located with RS segments with same key range.
- Big columns (megabytes) are stored in individual files in the OS

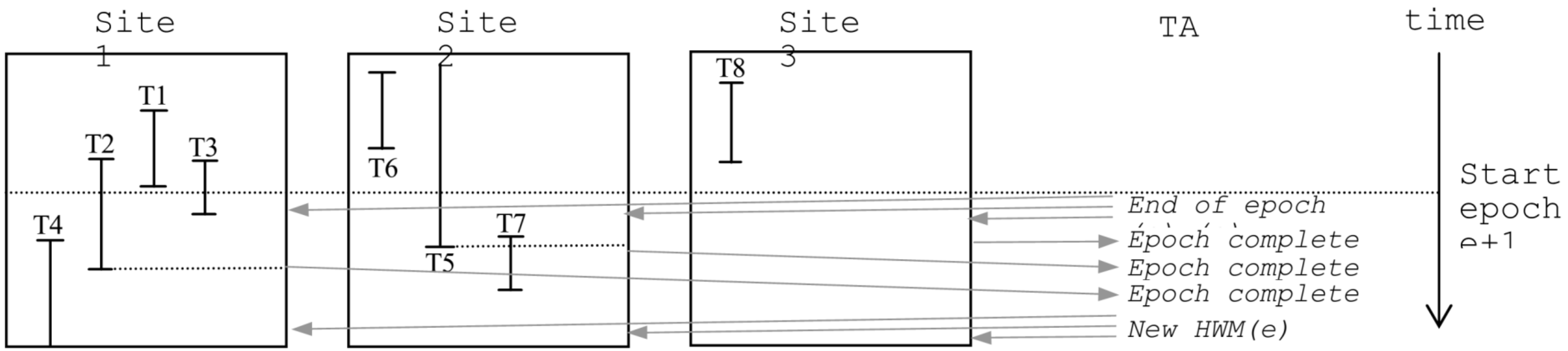
Updates and Transactions

Updates and Transactions

- Inserts are represented as new objects in WS.
  - SKs are globally unique, consistent between RS and WS
  - Keys in WS are consistent with RS storage keys.
- Very large main memory buffer pool
- Isolate read-only transactions using snapshot isolation
  - Accessing a consistent state in recent past
  - No locks
  - Timepoints: high water mark (HWM) and low water mark (LWM).
- Update transactions use two-phase locking

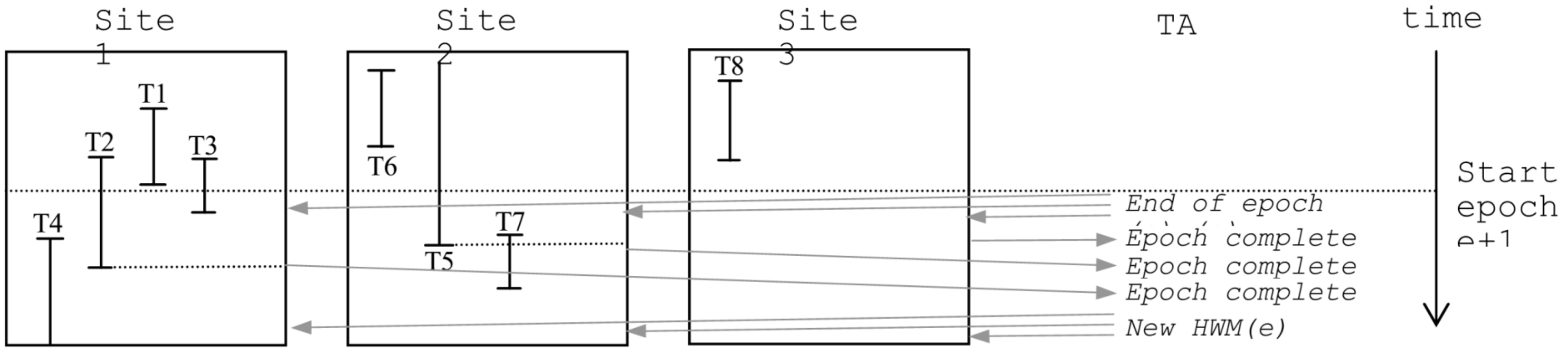
Updates and Transactions: Snapshot Isolation

- Determining what records are visible at effective time ET
  - An update is converted to an insert and a delete.
  - Coarse time granularity “epochs”.
  - Insertion vector (IV) for each segment in WS, recording epoch.
  - Deleted record vector (DRV) for each projection, stored in WS.
- Maintain High Water Mark (HWM)



Updates and Transactions: Snapshot Isolation

- Maintain High Water Mark (HWM)
  - Once TA broadcasts HWM with value e, read-only transactions can read data from e or earlier.
  - Epochs can be wrapped up.



Updates and Transactions: Locking-based Concurrency Control

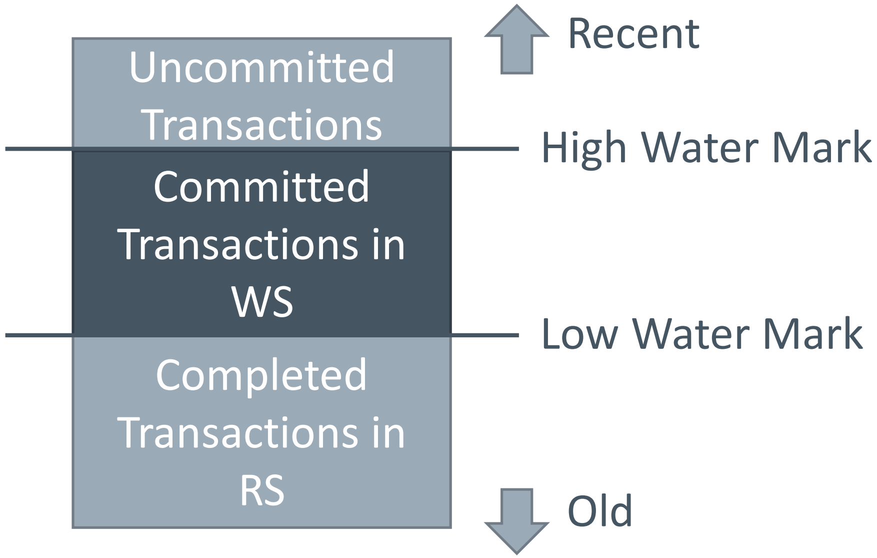
- Read-write transactions use strict two-phase locking.
  - Write-ahead logging for recovery
  - Only log UNDO records, and do not use strict 2-phase commit
  - Resolve deadlock via timeouts
- Distributed COMMIT processing
  - Master assigns units of work to sites
  - No PREPARE messages
  - Master does not issue COMMIT until all workers complete current actions
- Transaction Rollback
  - When transaction aborted, use UNDO log.
  - Logical logging

Updates and Transactions: Recovery

- Three types of crashes:
  - Failed site suffered no data loss
  - Failure destroying both RS and WS
  - WS damaged but RS is intact (common)
- Recovering WS with RS intact
  - Example: a WS segment Sr of a projection with a sort key K on a recovering site r along with a collection of other projections which contain the sort key of Sr.
    Each WS segment S (on remote sites) contains all tuples inserted with timestamp > $t_{lastmove}(S)$, the latter denotes the most recent insertion time of any record in S's corresponding RS segment.

Updates and Transactions: Recovery

```
SELECT desired_fields,
       insertion_epoch,
       deletion_epoch
FROM recovery_segment
WHERE insertion_epoch > t_lastmove(Sr)
      AND insertion_epoch <= HWM
      AND deletion_epoch = 0
          OR deletion_epoch >= LWM
      AND sort_key in K
```

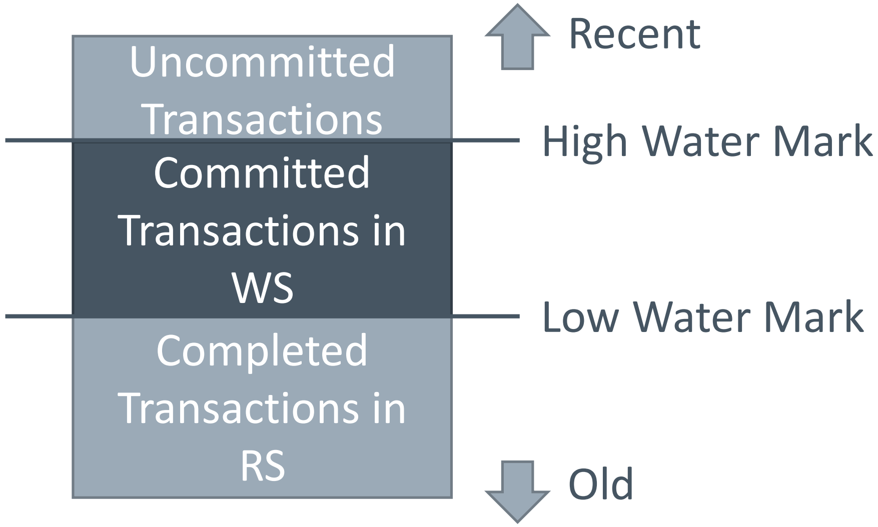


- Recovering WS with RS intact (continued)
  - If for all S in remote sites, there is $t_{lastmove}(S) \leq t_{lastmove}(Sr)$
    - Run the above queries
  - If $\exists S, such\ that\ t_{lastmove}(S) > t_{lastmove}(Sr)$, some of the tuples in Sr have already been moved to RS on the remote site
    - To handle this case, force the tuple mover to keep log. Use TM log and S to solve the above query.

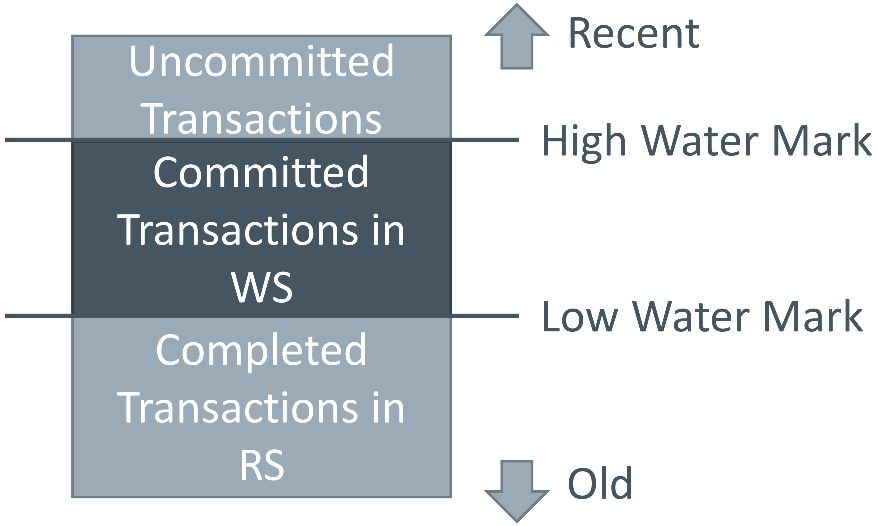
Tuple Mover

Tuple Mover

- Job: move blocks of tuples in a WS segment to the corresponding RS segment, and update join indexes.
  - Performs Merge-out Process (MOP) on the (RS, WS) segment pair
- MOP finds all records in the WS segment with an insertion time ≤ Lower Water Mark (LWM), and divides them into:
  - Ones deleted ≤ LWM. Discard them.
  - Ones not deleted; ones deleted after LWM. Move them to RS.



Tuple Mover



- MOP performs the operations by creating RS’ to replace RS
  - $t_{lastmove}$ is updated to the most recent insertion time in RS’, LWM is kept higher than or equal to it.
  - IV and DRV involved and maintained
  - LWM is updated periodically

Query Execution

Query Execution: Query Operators

- Accepts SQL query and constructs query plan. Major operators:
  - Decompress
  - Select: produces a bitstring representation of the normal select result.
  - Mask: accepts a bitstring B and projection Cs, restrict Cs by emitting only the values whose corresponding bits in B are 1.
  - Project
  - Sort
  - Aggregation Operators
  - Concat: combines one or more projections sorted in the same order into a single projection.
  - Permute
  - Join
  - Bitstring Operators

Query Execution: Query Optimization

- Selinger-style cost-based optimizer. But different from traditional query optimization
- 1. Need to consider compressed representations of data
  - An operator's execution cost is dependent on the compression type of input
  - Cost model must be sensitive to the representations of input and output columns
- 2. Decisions about which set of projections to use for a given query
  - Pruning the search space to find the plan
- 3. Decisions about when to mask a projection according to a bitstring

Performance Comparison

Performance Comparison

- RS and executor are ready, but not WS and tuple mover.
- Limited to read-only queries
- RS does not support segments or multiple grid nodes
- Benchmarking system: 3 Ghz Pentium, RedHat Linux, 2GB RAM
- Use a simplified TPC-H testing

```
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);
```

Performance Comparison

› 3 systems for comparison: C-Store, one row-store DBMS system and one column store DBMS system

– Turned off locking and logging

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

› Reason: compression and absence of padding, even with redundancy incorporated.

› Seven queries run on each system:

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
```

Performance Comparison

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
```

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
```

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
```

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

› Tuning parameters carefully selected, but may not be optimal

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

› Reasons for being fast:
  - Column representation avoid reading useless attributes
  - Stores overlapping projections, allows multiple orderings of a column
  - Better compression of data
  - Query operators operate on compressed representation

› Running the other systems with materialized views for "fair" comparison

	C-Store	Row Store	Column Store
Disk Usage	1.987 GB	11.900 GB	4.090 GB

Performance Comparison

- Running the other systems with materialized views for “fair” comparison (continued)
- C-Store 6.4 times faster than row-store with 1/6 space requirement; 16.5 times faster than commercial column-store with 1/1.83 space requirement.
- WS not mature. Incomplete comparison.

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

Conclusions

Conclusions

- A column store representation with the query execution engine
- A hybrid architecture which allows transactions on a column store
- Economizing the space cost by coding data values and dense-packing
- A data model consisting of overlapping projections of tables
- A design optimized for a shared-nothing machine environment
- Distributed transactions without a redo log or two-phase commit
- Efficient snapshot isolation

Q&A