CS 451

PROJECT 2

JOHN MOORE

TOPICS

01 L-STORE BACKGROUND

02 DATA MODEL IMPLEMENTATION

03 BUFFERPOOL MANAGEMENT

04 QUERY INTERFACE

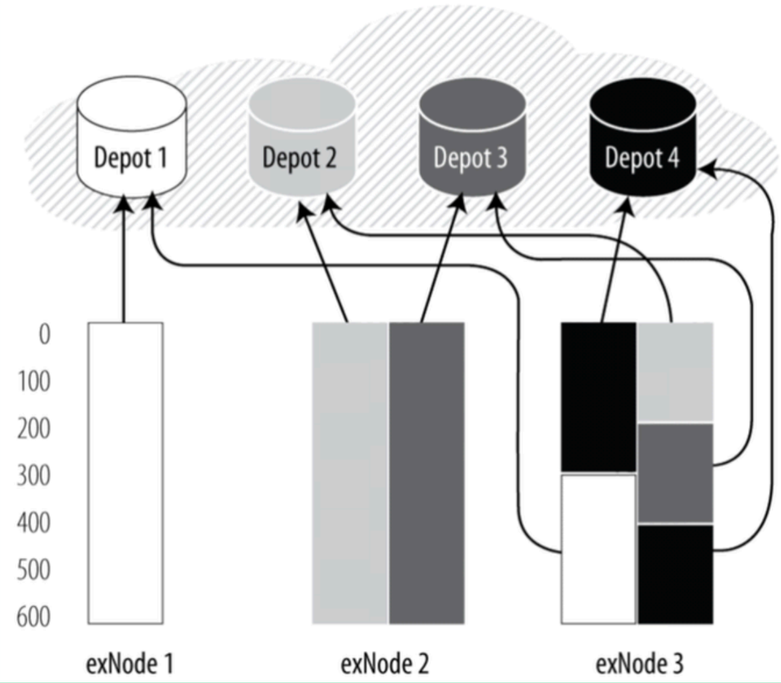
05 IMPLEMENTATION

06 LIVE DEMO SCRIPT

07 TECHNICAL CHALLENGES

L-STORE BACKGROUND

L-Store Vizualization



- L-Store: Unified OLTP & OLAP Engine
- Novel lineage-based storage architecture
- combines real-time transactional & analytical processing
- Efficient for write-optimized (tail) and read-optimized (base)
- Maintains current & historical data with consistency

L-STORE BACKGROUND

- Table structure combines base pages and tail pages
- Base pages: Store current record values
- Tail pages: Maintain version history
- Record metadata includes:
- Indirection: Links to previous versions
- RID: Unique record identifier

DATA MODEL IMPLEMENTATION

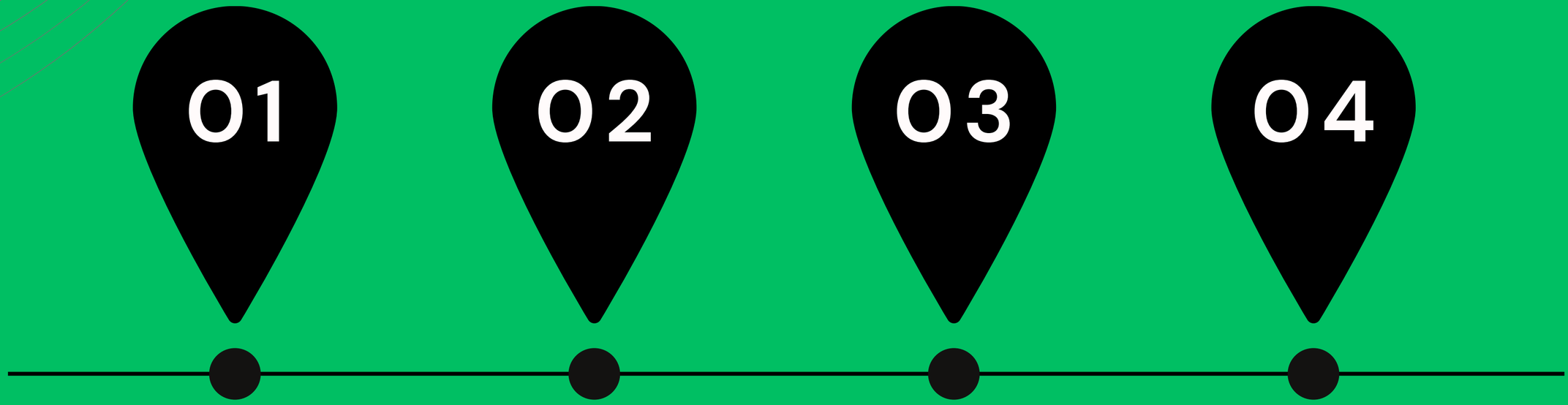
- Page-oriented storage with 4KB fixed size
- Lazy loading approach:
- Pages created dynamically as needed
- 8-byte integer slots (512 records per page)
- Columnar storage format
- Avoids loading unnecessary columns

BUFFERPOOL MANAGEMENT

- Select: Current and versioned records
- Insert: New record creation
- Update: Version chain maintenance
- Delete: Logical deletion
- Sum: Range-based aggregation
- Index-assisted lookups:
- Primary key indexing (default)
- Optional secondary indices
- Range query support
- RID-based record access

QUERY INTERFACE

DATA MODEL IMPLEMENTATION



COLUMN STORAGE APPROACH

BASE PAGES

TAIL PAGES

PHYSICAL PAGE STRUCTURE

BASE AND TAIL RECORD STRUCTURE

Record identifiers combine a unique counter with page location, managed in a page directory that maps RIDs to (page_type, page_index, slot) tuples for O(1) record access.

RID IMPLEMENTATION

A metadata column that creates a linked-list of record versions by storing pointers (RIDs) to previous versions, enabling version traversal from newest to oldest updates.

INDIRECTION COLUMN

A bitmap where each bit represents a column in the record - set to 1 if the column was modified in this version, enabling quick identification of which values changed in each update operation.

SCHEMA ENCODING

BUFFERPOOL MANAGEMENT

IN-MEMORY DATA ORGANIZATION

- Data organized in 4KB pages (512 integer slots per page)
- Columnar storage with separate arrays for base and tail pages
- Each column (data + metadata) gets its own page list for efficient access

PAGE DIRECTORY IMPLEMENTATION

- Hash table mapping RIDs to (page_type, page_index, slot) tuples
- Enables O(1) record lookup
- Distinguishes between base/tail/deleted records

RECORD LOCATION AND RETREIVAL

- Two-step lookup process:
  a. Use RID to find page location in directory
  b. Calculate exact byte offset: slot * 8 for 64-bit values
- Metadata columns (indirection, RID, timestamp, schema) stored similarly for consistency

QUERY INTERFACE

Efficiently allocates RIDs sequentially, writes to base pages with metadata, and maintains index - all in O(1)

CREATE & INSERT

Uses RID-based lookup through index for O(1) retrieval, with version chain traversal for latest/historical data

SELECT OPERATION

Creates new tail records for version history while keeping base records current, using bitwise schema encoding to track changes

UPDATE MECHANISM

QUERY INTERFACE (CONT)

Implemented logical deletion via page directory status change and index cleanup for consistent state

DELETE HANDLING

Leveraged index range scan and version awareness for accurate aggregation over key ranges

SUM AGGREGATION

- Hash-based indexing enables O(1) lookups
- Columnar storage optimizes analytical queries
- Version chain provides efficient historical access
- Buffered page management reduces I/O overhead

PERFORMANCE

IMPLEMENTATION HIGHLIGHTS

- *Columnar storage with separate base and tail pages*
- *Thread-safe design with proper locking mechanisms*
- *Smart version chain implementation using indirection columns*
- *Base record stays current while history moves to tail records*
- *Fixed 4KB page size with optimized slot calculations*
- *O(1) record lookups via page directory*
- *Hash-based indexing for quick access*
- *Memory-efficient storage using bytearrays*
- *Notable Features*
  - *Time travel queries (select_version)*
  - *Range-based operations*
  - *Column projection support*
  - *Support for aggregation queries*

LIVE DEMO

TECHNICAL CHALLENGES AND SOLUTIONS

Major Challenges

- *Version chain management was particularly complex, requiring careful pointer manipulation and state tracking*
- *Update operations needed to maintain both current state and historical versions without data loss*
- *Maintaining consistency between base/tail records during concurrent operations*

Solutions Implemented

- Robust Version Traversal
- Two-Phase Updates
- Efficient Schema Tracking
- The version system was particularly elegant because it reconstructed historical states by starting with current data and selectively "rolling back" only the columns that changed, rather than storing complete copies of each version.

TECHNICAL CHALLENGES AND SOLUTIONS

Lessons Learned:

Database Correctness

- *Small inconsistencies can cascade into major data issues*
- *Thorough validation at each step is critical*
- *Edge cases in version history need careful handling*

Debugging Practices

- *Detailed logging saved hours of troubleshooting*
- *Test scenarios must cover version history edge cases*
- *Systematic tracing of update chains is essential*

Lessons Learned:

Technical Insights

- *Understanding temporal database concepts is crucial*
- *Memory layout significantly impacts performance*
- *Complex pointer relationships require robust safeguards*

Design Principles

- *Simple solutions often outperform complex ones*
- *Proper metadata management is fundamental*
- *Atomicity in updates prevents data corruption*

EVALS

```
~/s/CS_45/CS451/Project1T/Project1 on main !5 ?5 ❯ python exam_tester_m1.py     at 06:26:32 PM
Table 'Grades' created successfully.
Insert finished
Select finished
Update finished
Aggregate finished

Statistics Summary:
  Insert Success Rate:    1000 / 1000 (100.00%)
  Select Success Rate:    1000 / 1000 (100.00%)
  Update Version 0 Success Rate:    1000 / 1000 (100.00%)
  Update Version -1 Success Rate:   1000 / 1000 (100.00%)
  Update Version -2 Success Rate:   1000 / 1000 (100.00%)
  Aggregate Success Rate: 1500 / 1500 (100.00%)
~/s/CS_45/CS451/Project1T/Project1 on main !5 ?12 ❯                took 5s at 06:26:39 PM
```

```
Table 'Grades' created successfully.
Insert finished
Select finished
Update finished
Aggregate finished

Statistics Summary:
  Insert Success Rate:    1000 / 1000 (100.00%)
  Select Success Rate:    1000 / 1000 (100.00%)
  Update Success Rate:    3000 / 1000 (300.00%)
  Aggregate Success Rate: 500 / 500 (100.00%)
~/s/CS_45/CS451/Project1T/Project1 on main !5 ?12 ❯
```

WORK BREAKDOWN

THANK YOU