L-Store: Lineage-based Storage Architectures

ECS165A: Winter 2023

Slides are adopted from Sadoghi, et al.  

*L-Store: A Real-time OLTP and OLAP System, EDBT’18*
Data Management at Macroscale: The Four V’s of Big Data
Data Management at Macroscale: The Four V’s of Big Data

- Anthem BlueCross BlueShield
- Walmart
- MasterCard
- FICO
- VISA
- IRS

John Doe
Data Management at Microscale: Volume & Velocity

Data Velocity

OLTP (Write-optimized)

Sales

Walmart

Motivations
OLTP (Write-optimized)

ECS165A
Data Management at Microscale: Volume & Velocity

Data Volume

OLAP (Read-optimized)

Data is Stale

OLTP (Write-optimized)

Extract-Transform-Load (ETL)

Data Velocity

Sales

Reports

Data is Stale
Data Management at Microscale: Volume & Velocity

OLAP (Read-optimized)

OLTP (Write-optimized)

Extract-Transform-Load (ETL)

Sales

Data Velocity

OLAP

OLTP

Walmart
One Size Does not Fit All As of 2012

Big Data Landscape

Copyright © 2012 Dave Feinleib
dave@vcdave.com
blogs.forbes.com/davefeinleib
One Size Does not Fit All As of 2017
Data Management at Microscale: Volume & Velocity

OLAP+OLTP (Read & Write-optimized)

Walmart
Storage Layout Conflict

Write-optimized (i.e., uncompressed & row-based) vs. read-optimized (i.e., compressed & column-based) layouts
Reducing Index maintenance: Velocity Dimension

Observed Trends

In the absence of in-place updates in operational multi-version databases, the cost of index maintenance becomes a major obstacle to cope with data velocity.
Reducing Index maintenance: Velocity Dimension

Observed Trends

In the absence of in-place updates in operational multi-version databases, the cost of index maintenance becomes a major obstacle to cope with data velocity.

Extending storage hierarchy (using fast non-volatile memory) with *an extra level of indirection* in order to
Reducing Index maintenance: Velocity Dimension

Observed Trends

In the absence of in-place updates in operational multi-version databases, the cost of index maintenance becomes a major obstacle to cope with data velocity.

Extending storage hierarchy (using fast non-volatile memory) with an extra level of indirection in order to Decouple Logical and Physical Locations of Records to Reduce Index Maintenance
Traditional Multi-version Indexing: Updating Records

Updating random leaf pages
Traditional Multi-version Indexing: Updating Records

Updating random leaf pages
Traditional Multi-version Indexing: Updating Records

Updating random leaf pages
Traditional Multi-version Indexing: Updating Records

Updating random leaf pages
Indirection Indexing: Updating Records

HDD

RID Index

RID Index

Eliminating random leaf-page updates
Indirection Indexing: Updating Records

 RID Index -> HDD -> RID Index

Eliminating random leaf-page updates
Indirection Indexing: Updating Records

LID: Logical Identifier
RID: Record Identifier

SSD
HDD

Indirection Index (LtoR Mapping)
Indirection Indexing: Updating Records

Eliminating random leaf-page updates

HDD

SSD

LID Index

RID: Record Identifier

LID: Logical Identifier

Tail (append-only)
Indirection Indexing: Updating Records

Eliminating random leaf-page updates

HDD
SSD
LID Index
RID: Record Identifier
LID: Logical Identifier
Tail (append-only)
Indirection Indexing: Updating Records

Eliminating random leaf-page updates

LID: Logical Identifier
RID: Record Identifier

HDD
Tail (append-only)

LID Index

SSD

Mohammad Sadoghi (UC Davis)
Observed Trends

In operational databases, there is a pressing need to close the gap between the write-optimized layout for OLTP (i.e., row-wise) and the read-optimized layout for OLAP (i.e., column-wise).
Unifying OLTP and OLAP: Velocity & Volume Dimensions

Observed Trends

In operational databases, there is a pressing need to close the gap between the write-optimized layout for OLTP (i.e., row-wise) and the read-optimized layout for OLAP (i.e., column-wise).

Introducing a *lineage-based storage architecture*, a contention-free update mechanism over a native columnar storage in order to
Observed Trends

In operational databases, there is a pressing need to close the gap between the write-optimized layout for OLTP (i.e., row-wise) and the read-optimized layout for OLAP (i.e., column-wise).

Introducing a *lineage-based storage architecture*, a contention-free update mechanism over a native columnar storage in order to lazily and independently stage stable data from a write-optimized layout (i.e., OLTP) into a read-optimized layout (i.e., OLAP).
Lineage-based Storage Architecture (LSA): Intuition

Physical Update Independence: De-coupling data & its updates (reconstruction via in-page lineage tracking and lineage mapping)
Lineage-based Storage Architecture (LSA): Intuition

Base Pages (Read-only)
Tail Pages (Append-only)
Index
Base Version (anchored RIDs)
Latest Version (monotonically increasing RIDs)
Append-only Updates (physical update independence)

Monotonically Increasing Lineage (updates are assigned RIDs in an increasing order)
Points to Stable RIDs (i.e., anchored RID)

In-page Lineage Tracking
Monotonically Increasing Lineage (updates are assigned RIDs in an increasing order)

Physical Update Independence: De-coupling data & its updates (reconstruction via in-page lineage tracking and lineage mapping)
Lineage-based Storage Architecture (LSA): Intuition

- **Base Pages (Read-only)**
- **Tail Pages (Append-only)**
- **Index**
- **Latest Version (monotonically increasing RIDs)**
- **Base Version (stable anchored RIDs)**
- **Append-only Updates (physical update independence)**
- **Lazy Update Consolidation** (snapshot reconstruction via lineage mapping & in-page tracking)
- **In-page Lineage Tacking**
  - Data Block RIDs Remain Unchanged (stable reference, anchored RIDs)
  - Monotonically Increasing Lineage (updates are assigned RIDs in an increasing order)
  - Points to Stable RIDs (i.e., anchored RID)

**Physical Update Independence**: De-coupling data & its updates (reconstruction via in-page lineage tracking and lineage mapping)
Overview of the lineage-based storage architecture

(base pages and tail pages are handled identically at the storage layer)
Records are range-partitioned and compressed into a set of ready-only **base pages** (accelerating analytical queries).
Recent updates for a range of records are clustered in their **tails pages** (transforming costly point updates into an amortized analytical-like query)
Recent updates for a range of records are clustered in their tails pages (transforming costly point updates into an amortized analytical-like query)
Recent updates are strictly appended, uncompressed in the pre-allocated space (eliminating the read/write contention)
L-Store: Detailed Design

Achieving (at most) 2-hop access to the latest version of any record (avoiding read performance deterioration for point queries)
L-Store: Detailed Design

Indirection Column (back pointer to the previous version)

Write Optimized (uncompressed, append-only updates)

New Version

Read Optimized (compressed, read-only pages)

Indirection Column (uncompressed, in-place update)

Achieving (at most) 2-hop access to the latest version of any record (avoiding read performance deterioration for point queries)
Achieving (at most) 2-hop access to the latest version of any record (avoiding read performance deterioration for point queries)
L-Store: Contention-free Merge

Contention-free merging of only stable data: read-only and committed data (no need to block on-going and new transactions)
L-Store: Contention-free Merge

Lazy independent merging of **base pages** with their corresponding **tail pages** (resembling a local left outer-join of the base and tail pages)
L-Store: Contention-free Merge

Indepedently tracking the lineage information within every page (no need to coordinate merges among different columns of the same records)
Contention-free page de-allocation using an epoch-based approach
(no need to drain the ongoing transactions)
Contenion-free page de-allocation using an epoch-based approach
(no need to drain the ongoing transactions)
L-Store: Epoch-based Contention-free De-allocation

Contention-free page de-allocation using an epoch-based approach (no need to drain the ongoing transactions)
L-Store: Epoch-based Contention-free De-allocation

- **Page Directory**
- **Indirection Column**

- **Read Optimized**
  - (compressed, read-only pages)

- **Write Optimized**
  - (uncompressed, append-only updates)

- **Asynchronous Lazy Merge**

**Contention-free page de-allocation using an epoch-based approach**
(no need to drain the ongoing transactions)
L-Store: Epoch-based Contention-free De-allocation

Contention-free page de-allocation using an epoch-based approach
(no need to drain the ongoing transactions)
Experimental Analysis
Experimental Settings

- **Hardware:**
  - 2 × 6-core Intel(R) Xeon(R) CPU E5-2430 @ 2.20GHz, 64GB, 15 MB L3 cache

- **Workload:** Extended Microsoft Hekaton Benchmark
  - Comparison with *In-place Update + History* and *Delta + Blocking Merge*
  - Effect of varying contention levels
  - Effect of varying the read/write ratio of short update transactions
  - Effect of merge frequency on scan
  - Effect of varying the number of short update vs. long read-only transactions
  - Effect of varying L-Store data layouts (row vs. columnar)
  - Effect of varying the percentage of columns read in point queries
  - Comparison with log-structured storage architecture (*LevelDB*)
Effect of Varying Contention Levels

Achieving up to $40\times$ as increasing the update contention
Effect of Merge Frequency on Scan Performance

Mixed OLTP + OLAP Workload; Low Contention
(1 Scan + 1 Merge Threads, Page Size = 32 KB)

Merge process is essential in maintaining efficient scan performance
Effect of Mixed Workloads: Update Performance

Eliminating latching & locking results in a substantial performance improvement
Effect of Mixed Workloads: Read Performance

Mixed OLTP + OLAP Workload; Medium Contention
(Total of 17 Threads + 1 Merge Thread, Page Size = 32 KB)

- Lineage-based Data Store (L-Store)
- In-place Update + History
- Delta + Blocking Merge

Coping with tens of update threads with a single merge thread
L-Store Key Contributions

- Unifying OLAP & OLTP by introducing lineage-based storage architecture (LSA)

- LSA is a native multi-version, columnar storage model that lazily & independently stages data from a write-optimized layout into a read-optimized one

- Contention-free merging of only stable data without blocking ongoing or incoming transactions

- Contention-free page de-allocation without draining ongoing transactions

- L-Store outperforms in-place update & delta approaches by factor of up to $8 \times$ on mixed OLTP/OLAP workloads and up to $40 \times$ on update-intensive workloads
Questions?
Thank you!

Exploratory Systems Lab (ExpoLab)
Website: https://expolab.org/