Modular Data Storage with Anvil

Mike Mammarella
Shant Hovsepian
Eddie Kohler
Motivation

• Data storage and databases drive modern applications
  • Facebook, Twitter, Google Mail, system logs, even Firefox
  • Yet hand-built data stores can outperform by 100x! [Boncz]

• **Changing the layout of stored data** can substantially improve performance
  • Recent systems implement custom storage engines

• Custom storage engines are hard to write
  • Reason: Must be consistent, fast for both reads and writes
  • What if you want to experiment with a new layout?
Can we give applications a simple and efficient modular framework, supporting a wide variety of different data layouts, enabling better performance?
Can we give applications a simple and efficient modular framework, supporting a wide variety of different data layouts, enabling better performance?

Yes we can!
Anvil

- Fine-grained modules called *dTables*
  - Composable to build complex data stores from *simple parts*
  - Easy to implement new dTables to store specialized data

- **Isolates all writing** to dedicated writable dTables
  - Many data storage layouts only add or change read-only dTables, which are significantly easier to implement
  - Good disk access characteristics come as well
  - Unifying dTables combine write- and read-optimized dTables
Contributions

- Fine-grained, modular dTable design
- Core dTables
  - Overlay dTable, Managed dTable, Exception dTable
- Anvil implementation
  - Shows that such a system can be fast
dTables

• Key/value store
  • Keys are integers, floats, strings, or blobs
  • Values are byte arrays
  • Iterators support in-order traversal
  • Most are read-only
dTables

• Key/value store
  • Keys are integers, floats, strings, or blobs
  • Values are byte arrays
  • Iterators support in-order traversal
  • Most are read-only

Slightly simplified, but not much!
dTables

• Key/value store
  • Keys are integers, floats, strings, or blobs
  • Values are byte arrays
  • Iterators support in-order traversal
  • Most are read-only

Slightly simplified, but not much!
dTable Layering

• Applications (and frontends) use the dTable interface
• But so do other dTables!
  • Transform data
  • Add indices
  • Construct complex functionality from simple pieces
dTable Layering

• Applications (and frontends) use the dTable interface
• But so do other dTables!
  • Transform data
  • Add indices
  • Construct complex functionality from simple pieces
dTable Layering

• Applications (and frontends) use the dTable interface
• But so do other dTables!
  • Transform data
  • Add indices
  • Construct complex functionality from simple pieces

![Diagram showing layering of dTables with lookup and iterator functions.](image-url)
dTable Layering

- Applications (and frontends) use the dTable interface
- But so do other dTables!
  - Transform data
  - Add indices
  - Construct complex functionality from simple pieces
An Application-Specific Backend

- Managed dTable
  - Journal dTable
  - Overlay dTable
  - Bloom dTable

- Exception dTable
  - State Dict. dTable
  - B-tree dTable

- Array dTable
- Linear dTable
An Application-Specific Backend

- Managed dTable
  - Journal dTable
  - Overlay dTable
  - Bloom dTable

- Exception dTable
  - State Dict. dTable
  - B-tree dTable
    - Array dTable
    - Linear dTable
Want to store the state of residence of customers
- Identified by mostly-contiguous IDs
- Most live in the US, but a few don’t
- Move between states occasionally

Common case could be stored efficiently as an array of state IDs
- But don’t want to penalize the uncommon case

Want transactional semantics
Application-Specific Data Example

- Want to store the state of residence of customers
  - Identified by mostly-contiguous IDs
  - Most live in the US, but a few don’t
  - Move between states occasionally

- Common case could be stored efficiently as an array of state IDs
  - But don’t want to penalize the uncommon case

- Want transactional semantics

- Mostly-contiguous IDs
- Most live in the US
- Some live elsewhere
- Don’t penalize them
- Occasionally relocate
• Stores an array of fixed-size values
  • Keys must be contiguous integers
  • Locating data items becomes constant time
  • Can’t store some types of data
  • Read-only
Storing Common Case Data Efficiently

- Managed dTable
  - Journal dTable
  - Overlay dTable
  - Bloom dTable
    - Mostly-contiguous IDs
    - Most live in the US
    - Some live elsewhere
    - Don’t penalize them
    - Occasionally relocate

- Exception dTable
  - State Dict. dTable
  - B-tree dTable

- Array dTable
- Linear dTable
Storing Common Case Data Efficiently

- Mostly-contiguous IDs
- Most live in the US
- Some live elsewhere
- Don’t penalize them
- Occasionally relocate
Storing Common Case Data Efficiently

Managed dTable

Journal dTable

Overlay dTable

Bloom dTable

Exception dTable

- Mostly-contiguous IDs
- Most live in the US
- Some live elsewhere
- Don’t penalize them
- Occasionally relocate

"California"

State Dict. dTable

Array dTable

31

B-tree dTable

Linear dTable
Storing Common Case Data Efficiently

- Managed dTable
  - Journal dTable
  - Overlay dTable
  - Bloom dTable
    - Mostly-contiguous IDs
    - Most live in the US
    - Some live elsewhere
    - Don’t penalize them
    - Occasionally relocate

- Exception dTable
  - “California”
    - State Dict. dTable
      - 31
      - Array dTable
  - B-tree dTable
    - Linear dTable
• Many data sets mostly but not entirely conform to some pattern that would allow more efficient storage
• Many data sets mostly but not entirely conform to some pattern that would allow more efficient storage

• Exception dTable combines a “restricted” dTable with an “unrestricted” dTable

• Sentinel value in restricted dTable indicates that the unrestricted dTable should be checked
• Many data sets mostly but not entirely conform to some pattern that would allow more efficient storage

• Exception dTable combines a “restricted” dTable with an “unrestricted” dTable

  • Sentinel value in restricted dTable indicates that the unrestricted dTable should be checked

• Simple unrestricted dTable: Linear dTable
Storing All Data

Managed dTable

Journal dTable   Overlay dTable   Bloom dTable

Exception dTable

State Dict. dTable   B-tree dTable

Array dTable   Linear dTable

- Mostly-contiguous IDs
- Most live in the US
- Some live elsewhere
- Don’t penalize them
- Occasionally relocate
Storing All Data

- Managed dTable
  - Journal dTable
  - Overlay dTable
  - Bloom dTable
    - Exception dTable
      - State Dict. dTable
      - B-tree dTable
        - Array dTable
        - Linear dTable

- Mostly-contiguous IDs
- Most live in the US
  - Some live elsewhere
  - Don't penalize them
  - Occasionally relocate
Storing All Data

Managed dTable

- Journal dTable
- Overlay dTable

Bloom dTable

Exception dTable

- State Dict. dTable
- Array dTable

- B-tree dTable
- Linear dTable

✔ Mostly-contiguous IDs
✔ Most live in the US
✔ Some live elsewhere
- Don’t penalize them
- Occasionally relocate
Storing All Data

- Managed dTable
  - Journal dTable
  - Overlay dTable
  - Bloom dTable
    - Exception dTable
      - State Dict. dTable
        - Array dTable
      - B-tree dTable
        - Linear dTable

- Mostly-contiguous IDs
- Most live in the US
- Some live elsewhere
  - Don’t penalize them
  - Occasionally relocate
Storing All Data

- Managed dTable
  - Journal dTable
  - Overlay dTable
  - Bloom dTable
    - Exception dTable
      - State Dict. dTable
        - Array dTable
      - B-tree dTable
        - Linear dTable

- Mostly-contiguous IDs
- Most live in the US
- Some live elsewhere
- Don’t penalize them
- Occasionally relocate
• We’ve seen how to build a read-only data store specialized for an application-specific layout
  • The pieces can be recombined for other layouts

• Next section shows how to build a writable store
  • Writable store dTables are common to many layouts
  • Split data write functionality and management policies
Writable dTables

• Array dTable is hard to update transactionally

• Idea: use separate writable dTables
  • Can be optimized for writing (e.g. a log)

• Several design questions
  • Implementation of write-optimized dTable
  • Building an efficient store from write-optimized and read-only pieces
Fundamental Writable dTable

- Appends new/updated data to a shared journal
Fundamental Writable dTable

- Appends new/updated data to a shared journal
Fundamental Writable dTable

- Appends new/updated data to a shared journal
- All data also cached in an AVL tree in RAM
Fundamental Writable dTable

- Appends new/updated data to a shared journal
- All data also cached in an AVL tree in RAM
- Should be “digested” when it gets large
System Journal

- Chronological order, append-only data store
  - Fast, contiguous writes on disks and other storage devices like Flash memory
  - Data later rewritten elsewhere in batches from cache

- *Clean* the system journal periodically to reclaim space
  - Data already written elsewhere can be omitted
  - Optimization: just delete it and restart if totally empty

- Uses a transaction system described in the paper
  - Client code chooses start and end of each transaction
  - Durability optional, consistency always provided
Handling Writes

Managed dTable

- Journal dTable
- Overlay dTable
- Bloom dTable

Exception dTable

- State Dict. dTable
- B-tree dTable

- Array dTable
- Linear dTable

- Mostly-contiguous IDs
- Most live in the US
- Some live elsewhere
- Don’t penalize them
- Occasionally relocate
Handling Writes

Managed dTable

Journal dTable

Overlay dTable

Bloom dTable

Exception dTable

✔ Mostly-contiguous IDs
✔ Most live in the US
✔ Some live elsewhere
✔ Don’t penalize them
- Occasionally relocate

State Dict. dTable

B-tree dTable

Array dTable

Linear dTable
Combining dTables

- Have: write-optimized and read-only dTables
- Want: one dTable that gives the best of both worlds
Combining dTables

- Have: write-optimized and read-only dTable
- Want: one dTable that gives the best of both worlds
- Idea: layer multiple read-only dTables together
  - Older data “lower” and newer data “higher”
  - Use a (writable) journal dTable “on top”
Combining dTables

- Have: write-optimized and read-only dTables
- Want: one dTable that gives the best of both worlds
- Idea: layer multiple read-only dTables together
  - Older data “lower” and newer data “higher”
  - Use a (writable) journal dTable “on top”
Combining dTables

• Have: write-optimized and read-only dTables
• Want: one dTable that gives the best of both worlds
• Idea: layer multiple read-only dTables together
  • Older data “lower” and newer data “higher”
  • Use a (writable) journal dTable “on top”
Combining dTables

- Have: write-optimized and read-only dTables
- Want: one dTable that gives the best of both worlds
- Idea: **layer multiple read-only dTables together**
  - Older data “lower” and newer data “higher”
  - Use a (writable) journal dTable “on top”
Combining dTables

- Have: write-optimized and read-only dTables
- Want: one dTable that gives the best of both worlds
- Idea: layer multiple read-only dTables together
  - Older data “lower” and newer data “higher”
  - Use a (writable) journal dTable “on top”
Unified View

- Managed dTable
  - Journal dTable
  - Overlay dTable
  - Bloom dTable
    - Mostly-contiguous IDs
    - Most live in the US
    - Some live elsewhere
    - Don’t penalize them
      - Occasionally relocate
  - Exception dTable
    - State Dict. dTable
    - B-tree dTable
      - Array dTable
      - Linear dTable
Unified View

Managed dTable

Journal dTable  Overlay dTable  Bloom dTable

Exception dTable

State Dict. dTable  B-tree dTable

Array dTable  Linear dTable

- Mostly-contiguous IDs
- Most live in the US
- Some live elsewhere
- Don’t penalize them
  - Occasionally relocate
Unified View

Managed dTable

Journal dTable

Overlay dTable

Bloom dTable

Exception dTable

State Dict. dTable

B-tree dTable

Array dTable

Linear dTable

✔ Mostly-contiguous IDs
✔ Most live in the US
✔ Some live elsewhere
✔ Don’t penalize them
✔ Occasionally relocate
Managed dTable

- Need a policy for digesting journal dTables
  - Decreases overlay performance, but frees memory
- Need a policy for combining read-only dTables
  - Restore overlay performance, consolidate data
- Must balance these goals efficiently
Managed dTable

- Need a policy for digesting journal dTables
  - Decreases overlay performance, but frees memory
- Need a policy for combining read-only dTables
  - Restore overlay performance, consolidate data
- Must balance these goals efficiently
Managed dTable

- Need a policy for digesting journal dTables
  - Decreases overlay performance, but frees memory
- Need a policy for combining read-only dTables
  - Restore overlay performance, consolidate data
- Must balance these goals efficiently
Managed dTable

- Need a policy for digesting journal dTables
  - Decreases overlay performance, but frees memory
- Need a policy for combining read-only dTables
  - Restore overlay performance, consolidate data
- Must balance these goals efficiently

---

Interfaces with transaction library
- Allows all other dTables to ignore transactions
Managing Long-Term Efficiency

Managed dTable
- Journal dTable
- Overlay dTable
- Bloom dTable
  - Exception dTable
    - State Dict. dTable
      - Array dTable
    - B-tree dTable
      - Linear dTable
Managing Long-Term Efficiency

- Managed dTable
  - Journal dTable
  - Overlay dTable
  - Bloom dTable
  - Exception dTable
    - State Dict. dTable
    - B-tree dTable
      - Array dTable
      - Linear dTable
Even with combining, we build up several overlaid read-only dTable subgraphs...
Even with combining, we build up several overlaid read-only dTable subgraphs...

Most of the data is probably in the older ones, combined from many others.
Bloom dTable

- Creates a Bloom filter for the keys in another dTable
  - Accelerates (most) nonexistent key lookups: O(1)!
  - Slightly slows down extant key lookups
  - Takes additional disk space in a separate file

- Read-only
  - No need to worry about key removal
  - Creates Bloom filter bitmap during create()

- Particularly useful under overlay dTables
An Application-Specific Backend

Managed dTable

- Journal dTable
- Overlay dTable
- Exception dTable

Exception dTable

- State Dict. dTable
- B-tree dTable

Bloom dTable

Array dTable

Linear dTable
An Application-Specific Backend

- Managed dTable
  - Journal dTable
  - Overlay dTable
  - Bloom dTable
- Exception dTable
  - State Dict. dTable
  - B-tree dTable
  - Array dTable
  - Linear dTable
## Additional dTables

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fixed-size</td>
<td>Combination array/linear</td>
</tr>
<tr>
<td>Unique-string</td>
<td>Deduplicates strings</td>
</tr>
<tr>
<td>Empty</td>
<td>Always empty</td>
</tr>
<tr>
<td>Memory</td>
<td>Not persistent</td>
</tr>
<tr>
<td>Cache</td>
<td>Memory cache</td>
</tr>
<tr>
<td>Small integer</td>
<td>Strips leading zero bytes</td>
</tr>
<tr>
<td>Delta integer</td>
<td>Stores differences</td>
</tr>
</tbody>
</table>
Performance Hypothesis

- Simple configuration changes can improve performance for specialized workloads
- Benefits of tailoring dTable configurations to data

- Performance is good for conventional workloads
  - Replaced SQLite’s update-in-place backend with Anvil
  - Can run a TPC-C-like benchmark (DBT2)

- Overhead of digesting and combining can be reduced by background processing
Evaluating dTable Modularity

- Load a given dTable configuration with 4M values
  - 0.2% of them 7 bytes, others 5 bytes
- Look up 2M random keys
dTable Choice Depends On Data

• Linear + B-tree vs. Array + Exception
  • Keys: contiguous or spaced 1000 apart

• Anvil’s modularity allows us to choose the right configuration for this data
Layered Index dTable Speeds Lookups

- Linear vs. Linear + B-tree
- Also measure time to create data store

- Usually a good configuration choice: many lookups will make up the create cost
Exception dTable Has Low Overhead

- Linear vs. Array vs. Array + Exception
- Plain array can store only fixed size values

- Exception dTable is low overhead vs. array (4% slower lookups here), but restores full functionality
How Does Read/Write Separation Perform?

- Anvil separates reads and writes into different dTables in our configurations.
- How does this perform relative to an update-in-place backend?
- Run DBT2 TPC-C with 1 warehouse for 15 minutes.
  - Simple row store Anvil configuration.
  - Digesting, combining, and system journal cleaning all set to occur frequently.
• Anvil’s durable configuration outperforms original durable configuration

• Anvil’s non-durable (but consistent, i.e. safe) configuration outperforms original “async” (i.e. unsafe) configuration
Better Disk Access Makes Anvil Fast

- Both Anvil configurations have significantly better disk access characteristics
- Larger, contiguous writes, better laid out on disk
- Can write *more* data in *less* time with faster seeks
Digesting and Combining

- Anvil’s performance benefits don’t come for free
  - Digesting, combining, and cleaning are the price

- These tasks can be done in the background
  - Read-only source data makes a background thread safe
  - Takes advantage of additional cores and spare I/O bandwidth

- Bulk loading a dTable with ~1GiB of data
  - Digest every few seconds
  - 50 seconds with background digest/combine
  - 82 seconds without
Related Work

- **Bigtable [Chang et al. ’06]**
  - Some aspects of Anvil resemble Bigtable SSTables
  - Write-optimized logs, read-optimized data
  - Higher-level distribution system complimentary

- **C-Store [Stonebraker et al. ’05]**
  - Data-specific optimizations and finer control of data layout

- **Abstraction-providing libraries**
  - Stasis transaction framework [Sears, Brewer ’06]
  - BerkeleyDB persistent data structure library
Conclusions

• Anvil provides a new way to build storage systems
• Desired functionality can be composed from fine-grained dTable modules
• Simple configuration changes allow storing data in many different useful ways
• Easy to write new dTables for novel storage strategies
Conclusions

• Anvil provides a new way to build storage systems
  • Desired functionality can be composed from fine-grained dTable modules
  • Simple configuration changes allow storing data in many different useful ways
  • Easy to write new dTables for novel storage strategies

• Still lacks some features, but they seem compatible
  • Aborting transactions, full concurrency
Conclusions

• Anvil provides a new way to build storage systems
  • Desired functionality can be composed from fine-grained dTable modules
  • Simple configuration changes allow storing data in many different useful ways
  • Easy to write new dTables for novel storage strategies

• Still lacks some features, but they seem compatible
  • Aborting transactions, full concurrency

• Performance overhead is small compared to potential benefits for applications
  • Prototype faster than SQLite’s B-trees for TPC-C