DATOM: A Proposal for an Alternative Storage System API

UNIVERSITY OF CAMBRIDGE

Calicrates Policroniades and Ian Pratt

Computer Laboratory, SRG

University of Cambridge

http://www.cl.cam.ac.uk

{cbp25, iap10}@cl.cam.ac.uk

Outline

- **Introduction**
  - The research problem and motivation: manipulation of structure and type.
- **The Storage System API**
  - Persistent Data Abstractions.
- **Model of Persistence**
- **Implementation**
  - High-level architecture and underlying mechanisms.
- **Evaluation**
- **Future Work and Conclusions**

The Research Problem

Do current storage technologies provide adequate support to manipulate data rich in structure and type?

- **File Systems (FS)**: Flat storage space and an API to operate on arrays of bytes [Daley and Neumann, 1965; Sandberg, 1986; Nagar, 1997].

↓

- **Relational Databases**: Tabular data representation, query capabilities, ACID transactional support [Codd, 1970].

↓

- **Object Oriented DB**: Well-known relational databases capabilities (query and transactions) + object oriented data model [Atkinson et al., 1989; Stonebraker et al., 1990].

↓

- **Persistent Programming Languages (PPLs)**: Merge the programming language and the data store into one system at runtime [Dearle, 1989; Atkinson, 1995].

✓ **<u>Programmers benefit with higher levels of abstraction</u>**: File Systems vs. PPLs.

What's the problem with current storage systems APIs?

- **Unbalanced trade-off between I/O efficiency and programmability in FS** [Gribble et al., 2000; MacCormick et al., 2004; OLE]:

  ➔ Considerable amount of data rich in type and structure (MPEG, PDF, HTML, XML JAR, TAR, soffice, etc.) or amenable to structural decomposition.

  ➔ Lack of ability to manipulate any abstraction: ***tedious and prone to errors!***

- **If applications are migrated to other storage technologies**:

  ✕ Addition of overheads: Intermediate language to access data (SQL, OQL, XQuery), transactional frameworks (ACID and long transactions), or complex data models.

  ✕ Loss of interoperability: Orthogonal persistence confines type support to a specific language compiler and adoption of a programming model.

- ***Mismatch* with applications' functional requirements**: Data-centric approach, well-defined access patterns with varying recoverability and consistency requirements.

Our Proposal

- **Depart from the flat file paradigm ➔ more abstract data representation.**

- **Creation of an efficient yet general storage system API for application data rich in structure and type.**
  - ➔ Retain a reasonable amount of structure and expose persistent data type.
  - ➔ Based on semantically rich and general abstractions.
  - ➔ Common use in applications code: Map, List, Matrix, Queue, and Stack.

- **Potential for impact:**
  - ✓ ***Less effort to develop*** persistence code: augment the level of abstraction and software quality.
  - ✓ Advanced ***data access strategies***: data prefetching, concurrency, and data sharing.
  - ✓ Assertive ***hints*** to persistent data access patterns.

Persistent Data Abstractions

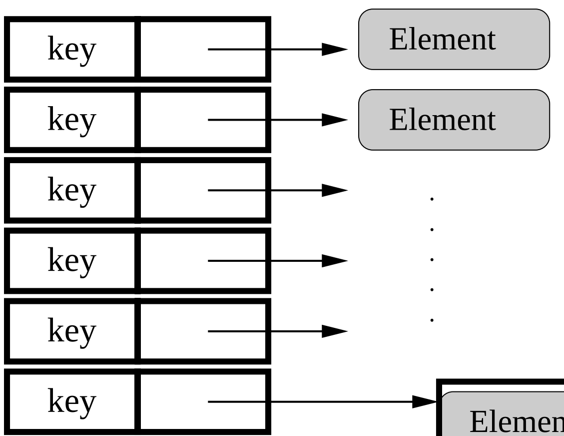
- ***Composite Entities***: Aggregation of application-specific data Elements.
  - ➔ Choose the right interface according to data access requirements.
  - ➔ Popular programming abstractions: expressive power, predefined semantics, potential to be implemented efficiently.

+	Map: Store elements associated with a key.
+	List: Collection of items in which certain order has to be preserved.
+	Matrix: Bidirectional access to collections of items.
+	Queue: Collection of items with FIFO access semantics.
+	Stack: Collection of items with LIFO access semantics.

- ***Elements***: Information-hiding items [Keedy and Richards, 1982].
  - ➔ Defined using Datom Data Language (DDL).
  - ➔ Application-specific semantics and types.
  - ➔ Final data containers: they do not reference to other data items.

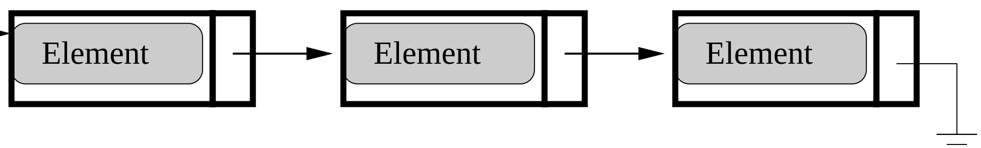
Example: Breaking File Data into Discernible Items

Composite Entity: Map




```
Interface Element: UserAddress {
   setStreet(String street)
   setCity(String city)
   getStreet()
   getCity()
   setLocationOnMap(URL url)
   setZipCode(String zc)
   ...
}
```

- Aggregation of CEs.
- Navigability.
- References in CE only.
- Elements for typed access.

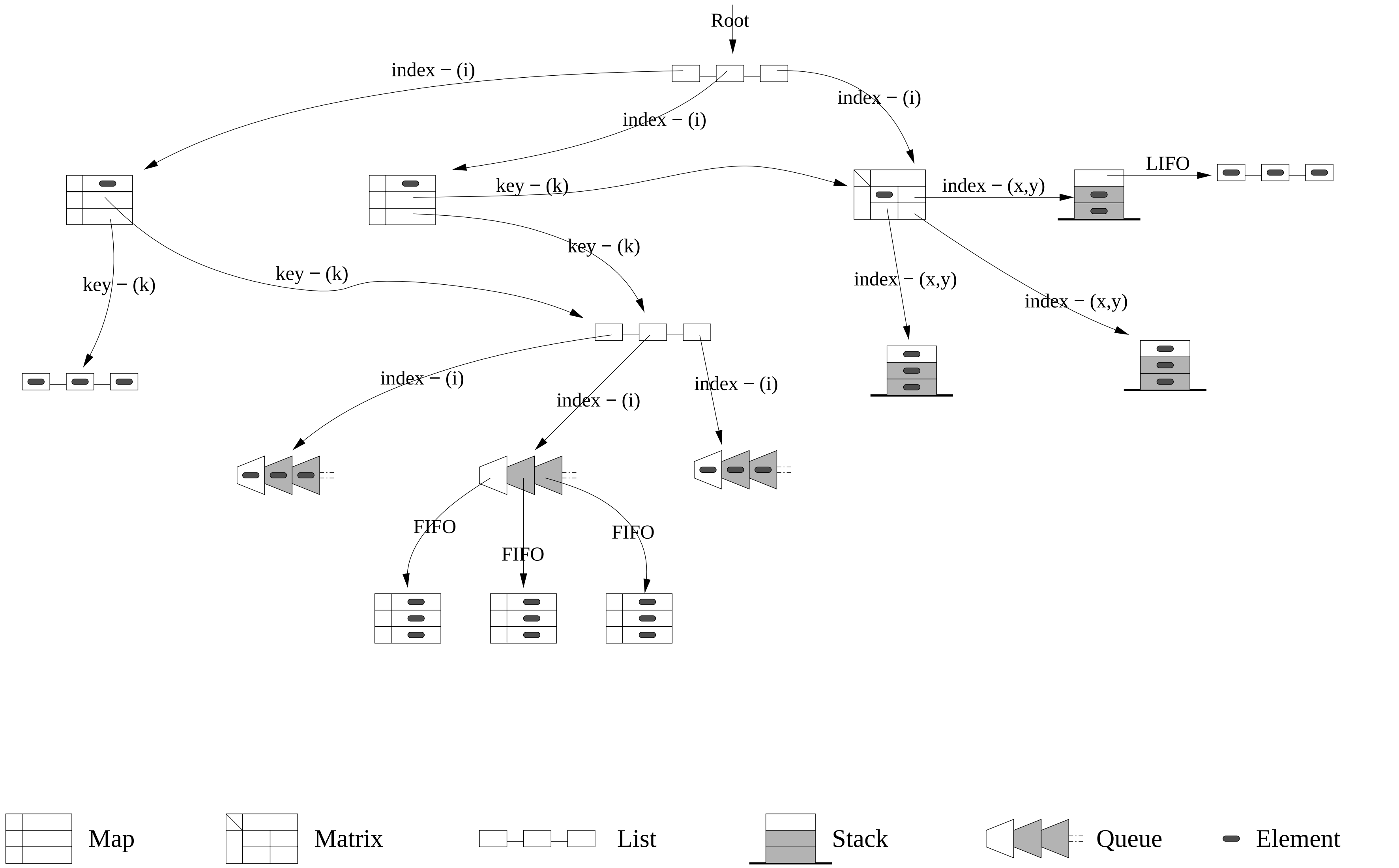



```
Interface Map {
   get(key)
   set(key, Element)
   del(key)
   length()
   merge(Map, override)
   haskey()
   ...
}
```

Composite Entity: List

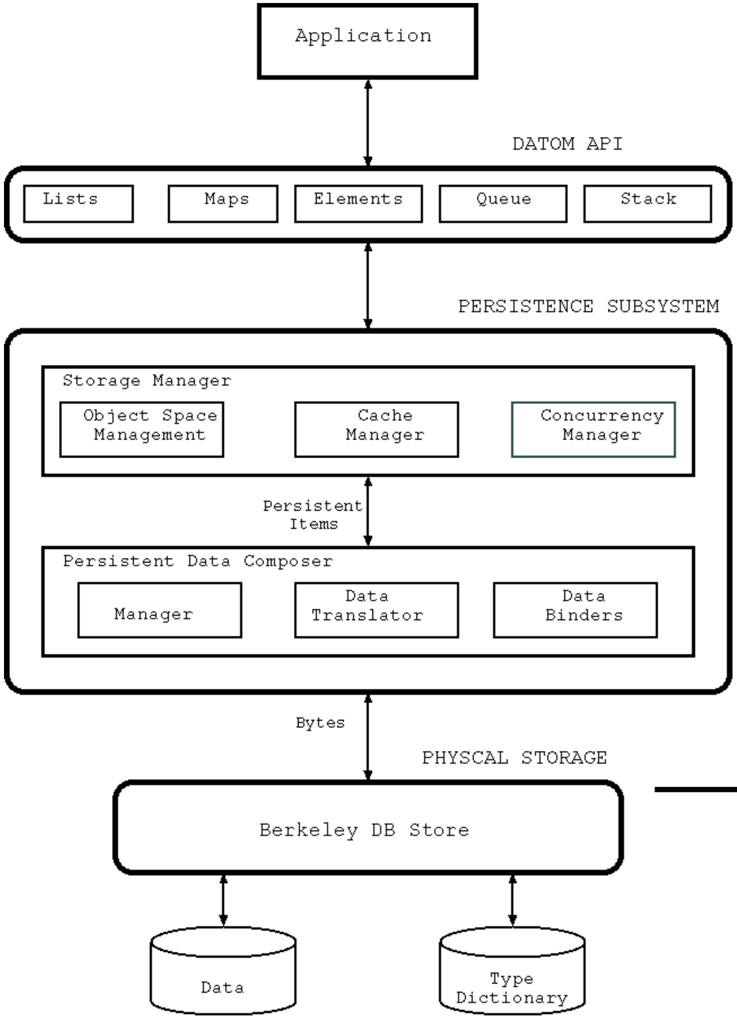
```
Interface List {
   add(Element)
   add(index, Element)
   get(index)
   remove(index)
   size()
   isEmpty()
   ...
}
```

Data Model


Root
index − (i)
index − (i)
index − (i)
key − (k)
LIFO
index − (x,y)
key − (k)
key − (k)
key − (k)
index − (x,y)
index − (x,y)
index − (i)
index − (i)
index − (i)
FIFO
FIFO
FIFO
Map
Matrix
List
Stack
Queue
Element

Implementation

High Level Architecture



→ Presents data abstractions and system functionality.

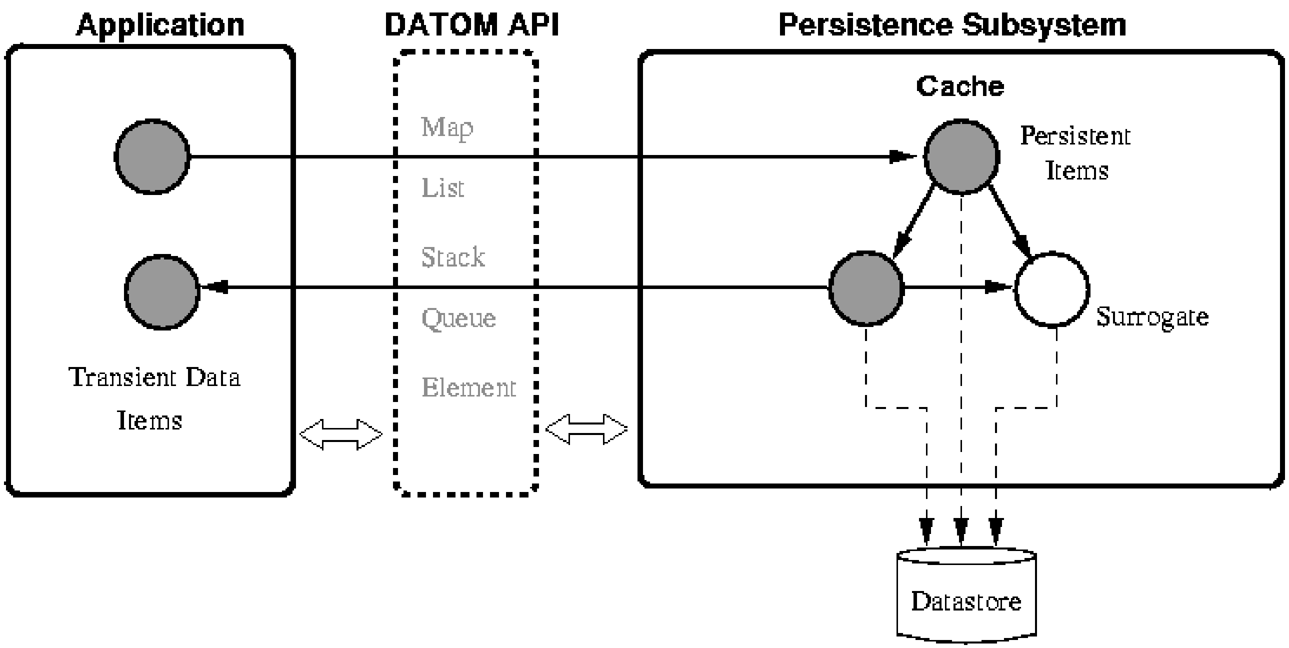
→ Creates, destroys, and moves persistent data:

- Items identity: lifelong logical IDs (PIDs).
- Memory management: Lazy loading and surrogates.
- Cache of persistent items.
- Concurrency control.
- Transform to/from physical storage format to run-time data representation.

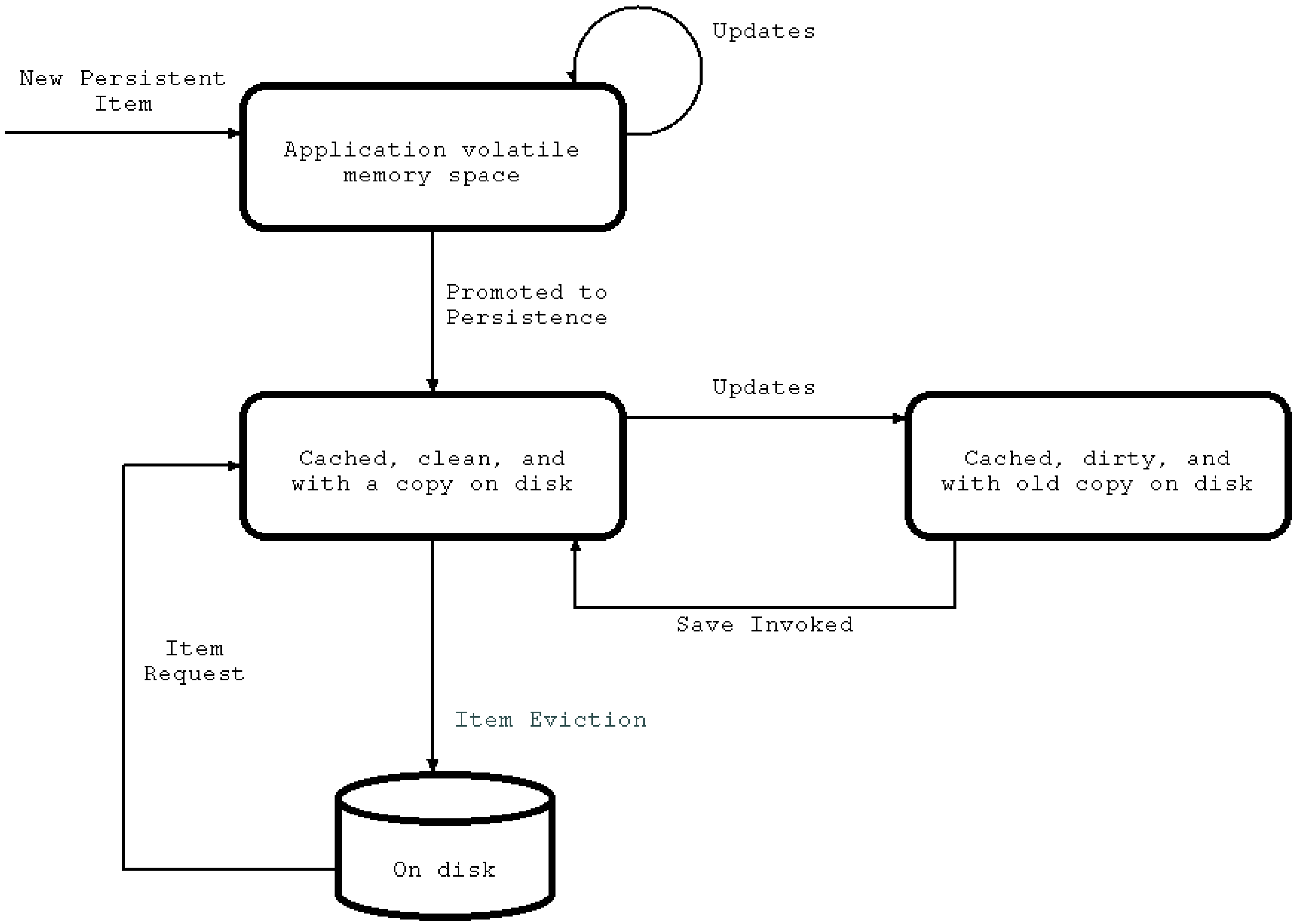
→ Atomically stores and fetches data as requested by the Storage Manager.

Model of Persistence

- **<u>Reachability AND Type:</u>** Smooth and complete control on data transferred to disk.
  - ➔ Any CE can be used as a root of persistence.
  - ➔ The system restricts by type the addition of *Elements* into the graph of persistence.
- **<u>Updates</u> are invoked from the roots of persistence: `CE.update();`**
  - ➔ Traversal of persistence graph: promotion of new items, and update of mutated items.
  - ➔ Persistent items exist in apps' memory space until promoted to persistence.



Persistent Data Life Cycle



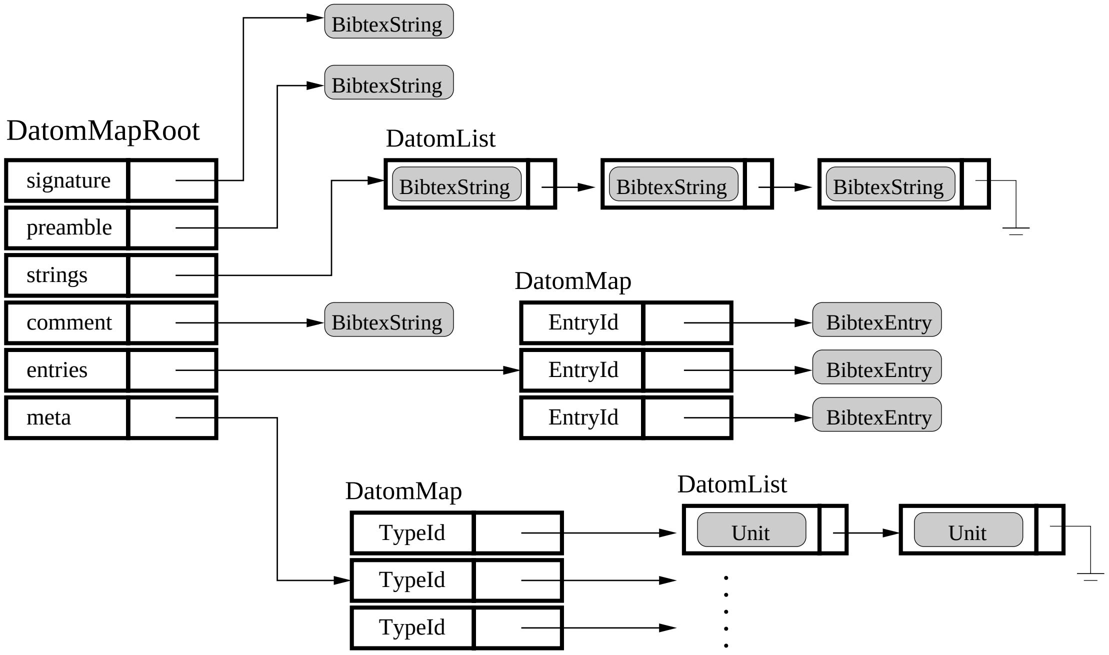
Results

- **Porting applications on top of the Datom API: Bibkeeper.**
  - Application recovered file data to a graph of persistent objects.
  - Remove parsing and serialization libraries.
  - Made the code self-explaining.
  - Potential avoidance of redundant data transmission to disk.
  - Application size reduced.

```
// Connecting to the store
StoreConn myStore = new StoreConn(cfgObj);

// Getting a root map
RootDatomMap myRefs = myStore.open(“bibtexRefs”);

// Getting an application data Element
Reference ref = (Reference) myRefs.get(“gray:1998”);

// In-memory updates
ref.setTitle(“Transaction Processing: Concepts and Techniques”);
     ref.setYear(1998);

// Pushing changes to stable storage as an atomic operation
myRefs.save();
```

Bibkeeper: The graph of persistence.

1) Map, List, and different types of Elements.
2) Changes in the morphology of the application.



Results

- **Bibkeeper: Source code measurements.**
  - PCMT tool [Gri97].
  - It collects metrics related with lines of code and classes that contain persistent code.
  - Parsing and tracking lines of code as productions rather than textual text.

Version	LoCs	PLoCs	# Classes	# Persistent Classes
File-based	6002	208	81	15
Datom-based	5570	469	76	40

- **Key findings.**
  - Reduction in size: code (432) and classes (5).
  - PLoCs and # of persistent classes increased: Explicit tracking of persistent objects.
  - Breakdown per class shows: programs either modify persistent abstractions all over source files; or show high locality => directly related to density of PLoCs.

Results – CDC Framework

- **<u>Cognitive Dimensions Framework</u>**.

➔ Usability aspects of the API contrasted with cognitive demands of different programming styles: Opportunistic, Pragmatic, and Systematic.

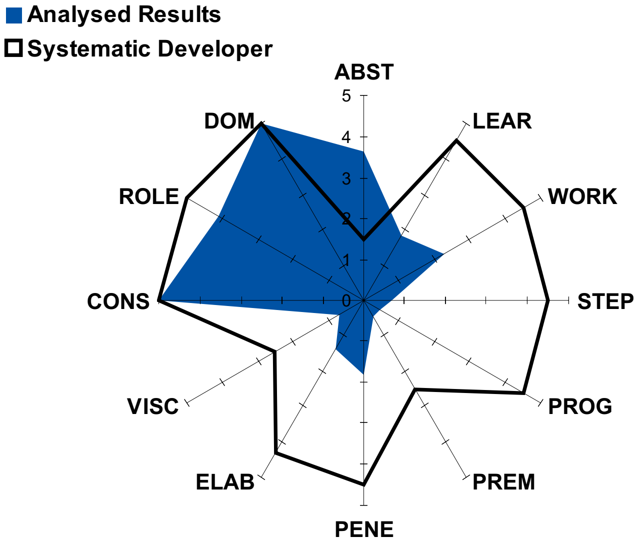
➔12 dimensions evaluated through:

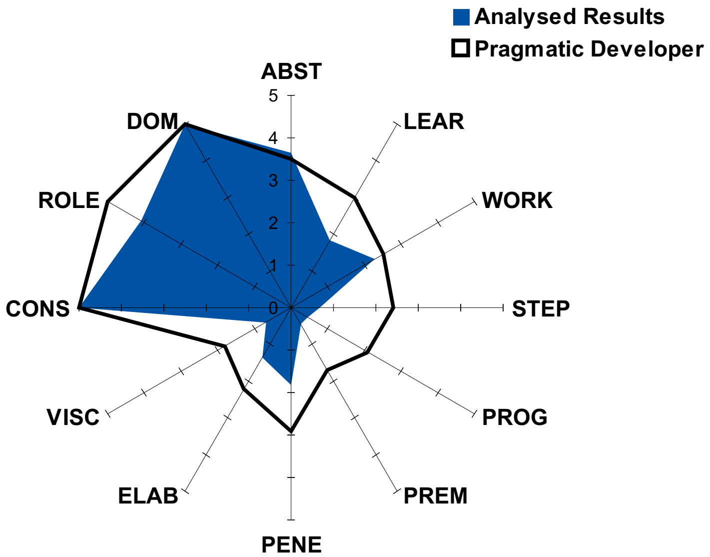
➔ Task analysis: Typical use scenarios.

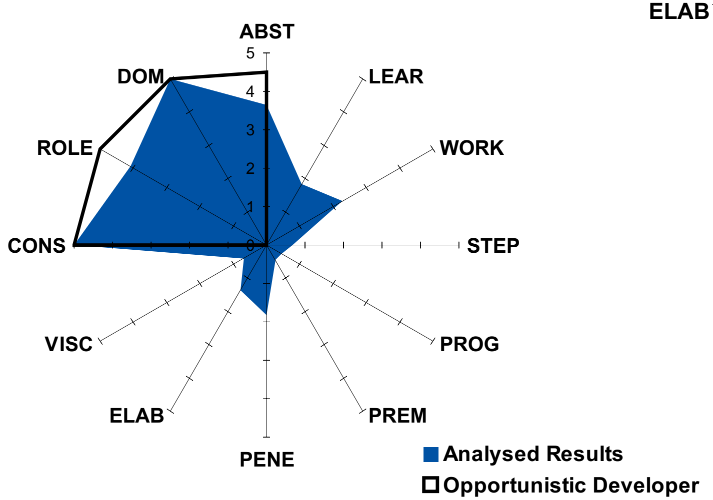
➔Code snippets for each main task.

1.	Recover a root of persistence.
2.	Setup a graph of persistence
3.	Add an Element to the graph of persistence.
4.	Update an Element.
5.	Read data from an Element.
6.	Delete an Element.
7.	Remove a Composite Entity.
8.	Apply atomic updates.
9.	Modify the morphology of the graph of persistence.
10.	Query the graph of persistence.
11.	Navigate and update the graph of persistence.

Results – CDC Framework







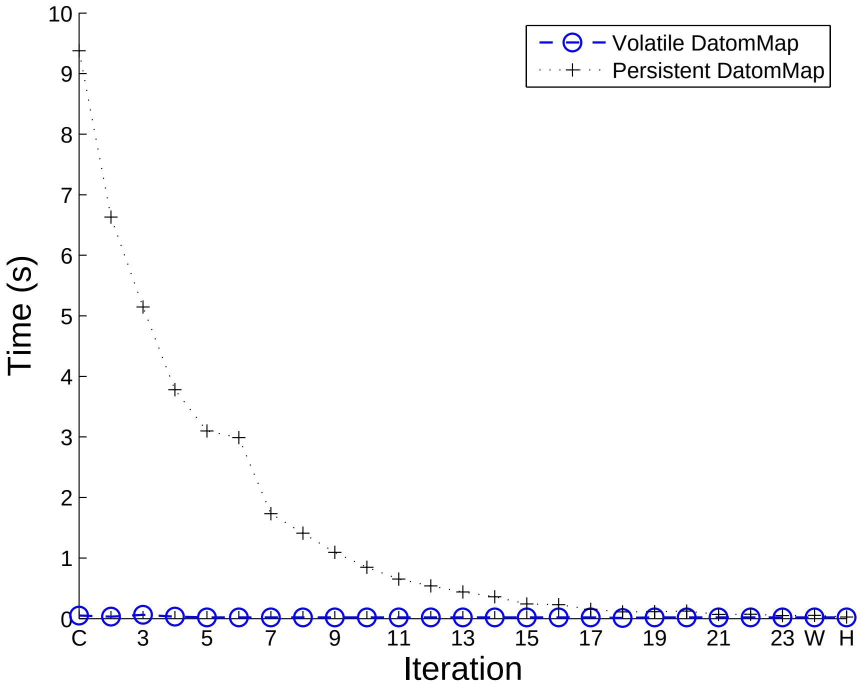
Results – Performance

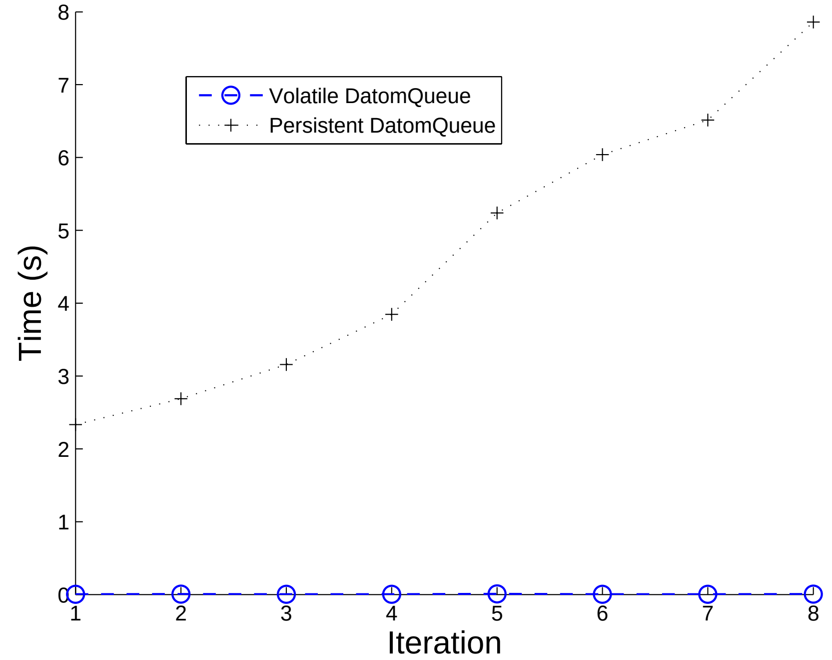
- **Datasets:** 36,000 Elements of type Data (int, int, int, double, String, Data).

- **Read barrier: Persistent item faulting.**
  - <u>Selective retrieval</u>: Map, List, and Matrix.
    - Run: 25 iterations, 9000 Elements per iteration. Simple read procedure. Measures cache warming.
  - One-way retrieval: Stack, and Queue.
    - Run: 8 iterations, 1000(n) Elements per iteration, Simple read procedure.

- **Write barrier: Detecting and logging updates.**
  - Similar access strategies but and update procedure is applied on fetched Elements.

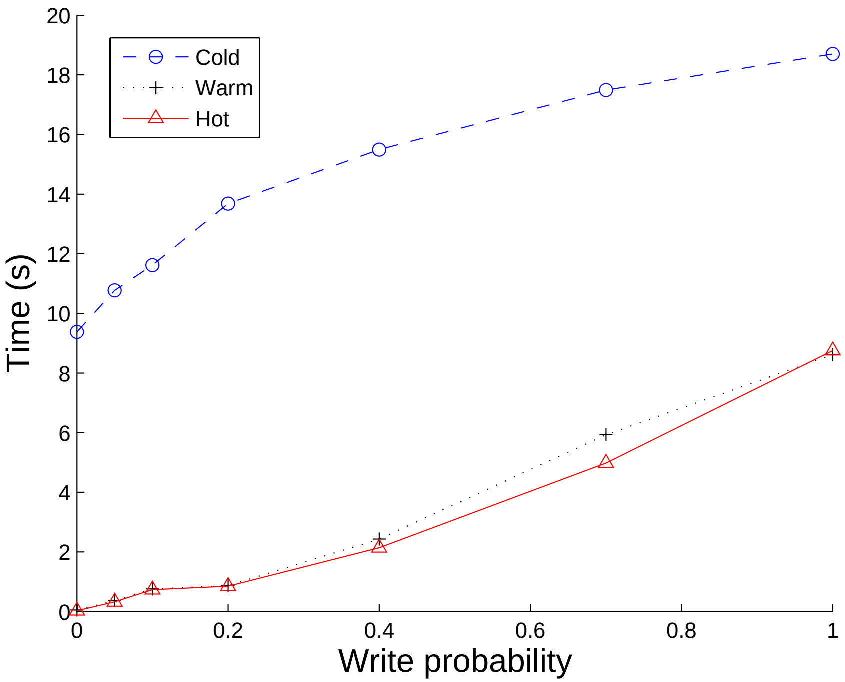
Results – Read barrier

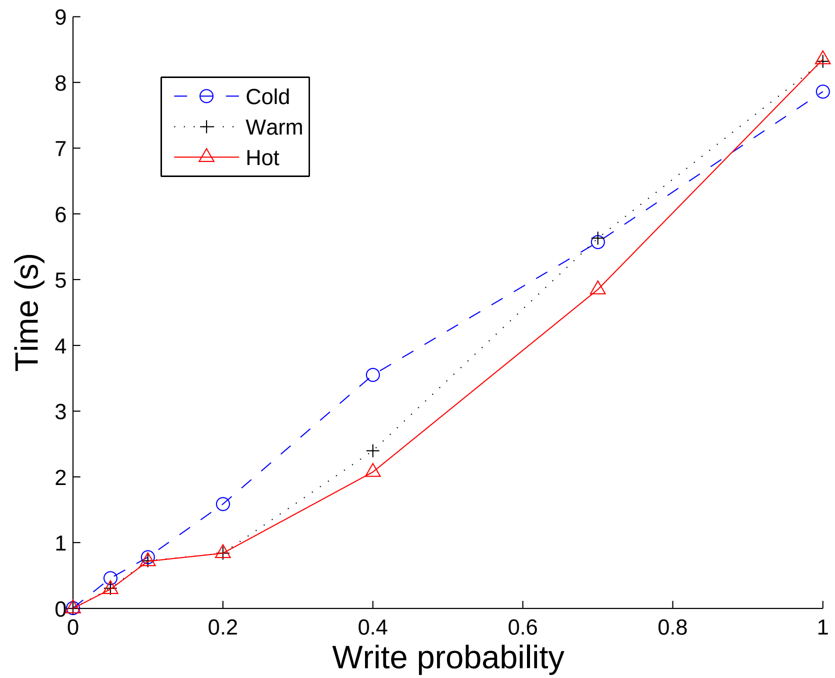




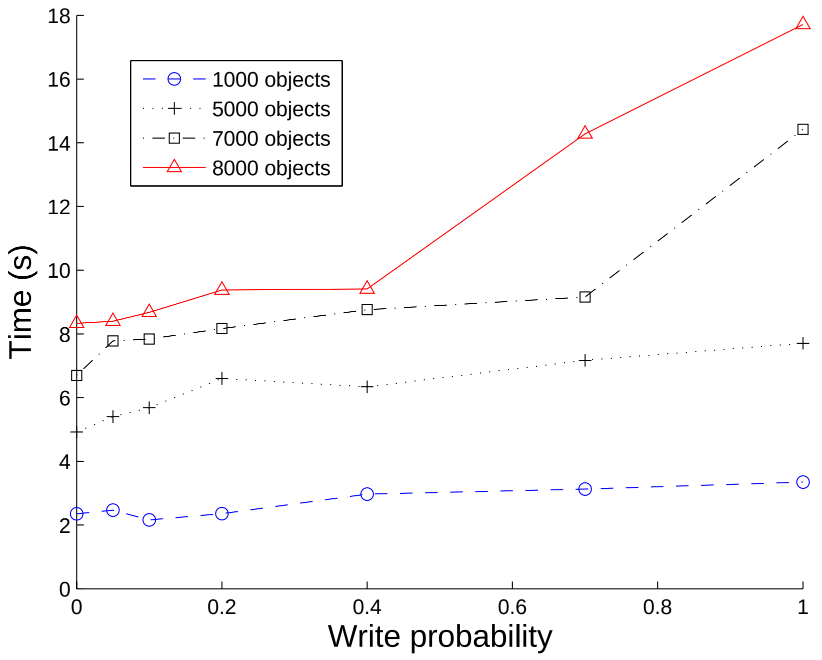
- **Read barrier: Persistent item faulting.**
  - Selective retrieval: Warming of caches.
  - One-way retrieval: Constant increase of time.

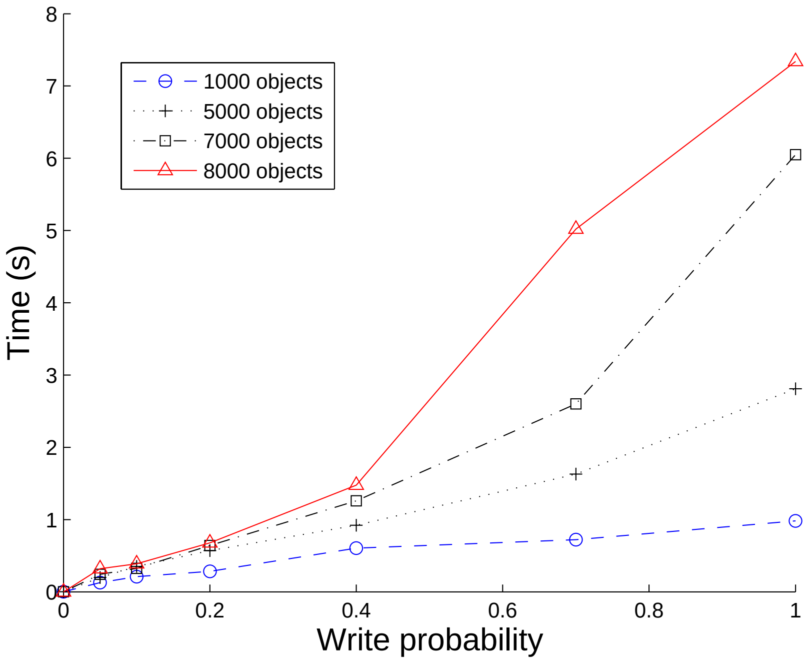
Results – Write barrier





- **Write barrier: Detecting and logging updates. Selective retrieval.**
  - Caches warming effects.
  - Checkpoint independent of cache state.





- **Write barrier: Detecting and logging updates. One way retrieval.**
  - Constant times according with the density of updates.

Future Work and Conclusions

- ***Work to be done***:
  - ➔Developers' feedback. http://www.cl.cam.ac.uk/~cbp25/datom/apidocs/
  - ➔Porting more applications.
  - ➔Ad-hoc storage layer to exploit abstractions.
  - ➔Partial checkpoints??

- ***Conclusions***:

  ➔ DATOM: A storage system whose API captures a judicious degree of structure and data type.

  ➔ Applications' persistent code can be simplified and developers' job eased.

  - Management of persistent data layouts and provision of data integrity services.
  - Sophisticated data access strategies based on applications' persistent data semantics: key, position, type, or content.
  - Fine-grained data manipulation to enable data sharing and concurrency.

The End

Questions?

Removing complexity

- ACID transaction for applications that need them and use more relaxed access semantics.
- Reachability of a set of well-known elements.
- Read and update barriers related to granularity of the objects, Elements as collections of small data items.

- Learning curve for programmers vs. DB models.
- A model to reason about.