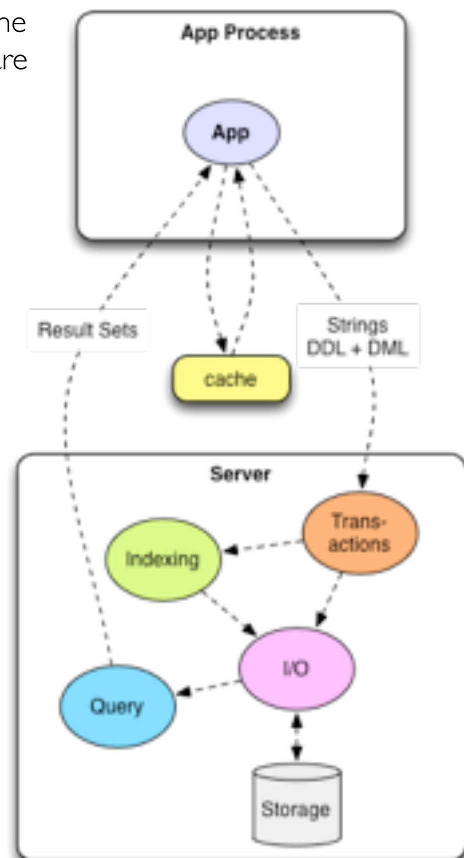
Alexander Kiel
University Leipzig
HAL8 Paper
02.05.2013

DATOMIC - A FUNCTIONAL DATABASE

Functional languages bought us persistent data structures [CO96] which are on its own a strong component of a safe and side effect free programming style. Even non-pure functional languages such as Clojure [CL] gain a lot of benefits from that data structures. Datomic [DA] brings persistent data structures into the durable world of databases. In Datomic the content of a whole database is one gigantic immutable value at one particular moment in time. Once a process has such a database value in hand, it can query it as long as it likes in perfect isolation to all other processes. Transactions create new values very time new facts become known to the database system.

Key Concepts of Datomic

Datomic is not an relational database and its not one of the typical NoSQL databases. In Datomic key database concepts are rethought in a way that key problems of traditional relational databases are addressed.



Traditional Database (from datomic.com)

Separating reads from writes

The first problem is that traditional databases require coordination for reads because they change data in place and so a read has to be able to see the old data as long as it takes, preventing the update to be carried out fully. That coordination around a fixed place of data is a big performance killer in database systems.

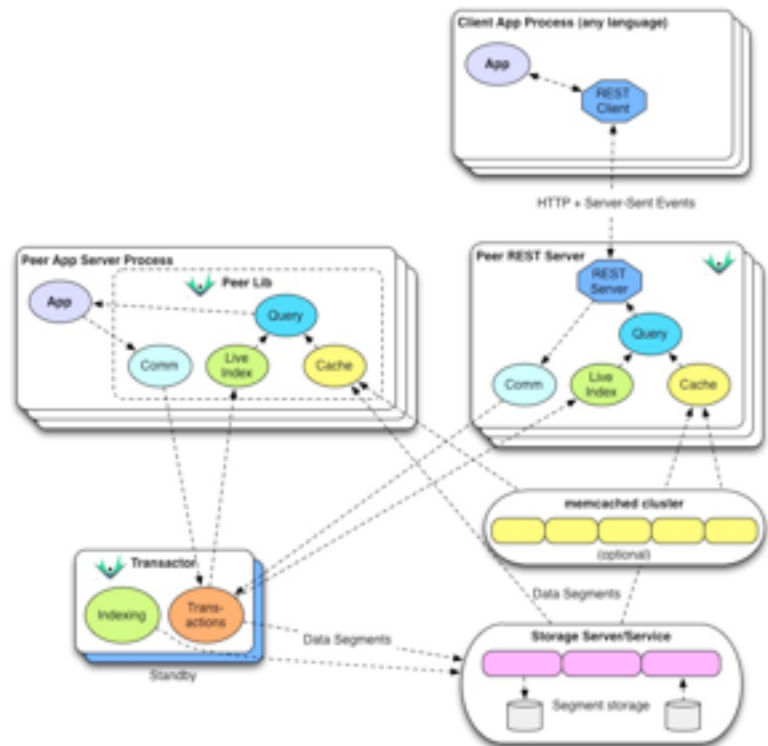
Datomic addresses this problem by using a persistent data structure for the whole database. With such a data structure in place, reads access one big immutable value which never changes and so don't require coordination with writes. Writes on the other hand produce new values without touching the old ones and so need no coordination with reads.

Taking the Architecture apart

The second problem of traditional databases lies in the architecture of the processes being part of the system. A traditional database consists of one server

process and many client processes. This architecture prevents the system to scale out. In fact only scaling up is possible which has its limits. The single server process has to do all the work which consists of performing queries, making updates and handling storage.

Datomic breaks this architecture apart by separating the query engine, the transaction handler and the storage engine into different processes.



Datomics Architecture (from datomic.com)

First the query engine moves into the client process as library. The so called Peer Lib knows how to access raw data segments from storage and does the query work on its own without the need of one central database server. As reads don't require coordination, a database system can now consist of as many query engines as there are clients and so the query side is arbitrarily scalable.

Second the writes go through a single transaction handler called Transactor. Writes need to be serialized in order to become atomic operations. As such the transactor is not scalable. Datomic takes the full ACID route into the database world and so writes can't be scalable either which is theoretically proven in the CAP theorem [LG02].

Third the storage is also separated into its own process because there are very scalable storages available which are most often NoSQL data stores. By being able to switch between different data stores a system can achieve different levels of read scaling and write throughput.

Time as First Class Concept

The third problem Datomic addresses, is that time is not a first class concept of relational databases. If the problem domain has questions like: "Who many customers had we at the end of 2012?" one has to write special queries which take time into account using custom timestamps in the data. In Datomic the very same query can be executed on every database value based on every time. Datomic achieves that by holding the history of all database values so that one is able to obtain an value from the end of 2012 and run the normal customer count query against it.

Basics of Persistent Data Structures

Persistent data structures are data structures which preserve its previous version whenever they are modified and thus are essentially immutable. This behavior contrasts traditional (ephemeral) data structures which are mutable.

Different Categories of Persistent Data Structures

There are three different categories of persistent data structures: partial persistent, fully persistent and confluently persistent data structures. Partial persistent data structures support the access of all versions but only allow the newest version to be modified. Fully persistent data structures also support modification of all versions. Confluently persistent data structures have additional meld operations on top of fully persistent data structures which somehow merge more than one version of such a data structure [DST94, FK01]. Examples for meld operations are list concatenation and set union.

Fully persistent data structures are the most common type used in functional programming languages. On the other hand Datomic is currently only partially persistent because one can only modify the newest version [1].

Implementation of Persistent Data Structures

While a naive implementation of a fully persistent data structure would simply copy its whole content on each modification (copy-on-write), practical implementations use structural sharing. The idea behind structural sharing is, that one can refer to unchanged parts of an older version instead of coping them. Such references are possible because the older versions are immutable of course. So one needs only to add the new or changed parts to the previous version to gain the new version.

The simplest fully persistent data structure is the singly linked list. Such lists consist of two constructors: *nil* and *cons*. The *nil* constructor takes no argument and represents the empty list whereas the cons constructor takes the two arguments *x* and *xs* and prepends the element *x* to the list *xs*. In this way the empty list is represented by *nil*, a list with one element *a* by *cons(a, nil)* and a list with the two elements *a* and *b* by *cons(a, cons(b, nil))*.

NIL empty list

A → NIL list with one element

A → B → NIL list with two elements

A simple implementation of the *cons* constructor which creates a node holding the element *x* and a pointer to the list *xs* leads directly to a fully persistent data structure. In the figure above such nodes are represented through green boxes holding an element and pointing to another list. Its easy to see that prepending the element *a* to the list *cons(b, nil)* doesn't change its previous version *cons(b, nil)*.

A → B → NIL

new version previous version

The implementation of more advanced data structures like sets, map and other forms of lists are out of scope of this introduction. Usually such data structures are implemented using trees.

Informational System of Facts

In Datomic data is stored in the form of facts. A fact is a statement which is true at a given point in time. An example may be the fact: "Sally likes pizza.". Such facts are called Datoms. Each Datom consists of four components: an entity, an attribute, a value and a point in time.

The table above shows the mapping of the components of the fact "Sally likes pizza." into those four categories. As such the data model of Datomic is similar to RDF Triple Stores with the additional fourth component of time.

Facts can be added to or retracted from the database. The time component is always taken from

Entity	Attribute	Value	Time
Sally	likes	pizza	02.05.2013

the time of the transaction in which a fact is added or retracted. So if Sally likes pizza from now on, one would add the fact "Sally likes pizza." to the database. If one day Sally doesn't like pizza anymore, one would retract the same fact from the database. If a fact is retracted, its not longer part of the information stored in the current database but older databases hold that fact forever.

Using facts of such a fine granularity, modifications of a database can be carried on in small steps which don't consume much storage space. Together with full support of transactions over arbitrary large sets of additions and retractions, one has the possibility to perform small and big changes in an efficient way while preserving atomicity. This model contrasts document stores or key-value stores which have only one granularity of atomic update, the whole document or the often large value.

Query Language

Datomic uses a form of Datalog as its default query language. Datalog is a subset of the logic programming language Prolog. In Datalog is a deductive query system which consists of a set of facts and rules which derive new facts from existing facts. Queries are partial specifications of facts or rules which return a set of all matching facts satisfying the specifications. A simple query finding all people which like pizza look like this in the Clojure syntax of Datomics Datalog:

In this syntax *?e* is a free variable in the partial specification *[?e :likes "pizza"]* which satisfies every entity there the fact ?e likes pizza is true. Querying a database consisting of the fact *[Sally :likes "pizza"]* returns a set containing Sally.

```
[:find ?e :where [?e :likes "pizza"]]
```

Joins are implicit in Datalog because every free variable has to satisfy the same value in all specifications. If one wants to query for all people which like fast food, one can write the following query:

In this query *?m* stands for a meal as pizza or solyanka. This query over a database consisting of the facts *[Sally :likes pizza]*, *[Tom :likes solyanka]* and *[pizza :is "fast-food"]* would return only Sally but not Tom. In the sense of relational databases and SQL this query is comparable to a join over a Person

```
[:find ?e :where [?e :likes ?m][?m :is "fast-food"]]
```

and a Meal table.

Rules are out of the scope of this paper but are comparable to views in SQL.

Summary

Datomic is an very interesting new database system which separates itself from traditional relational database systems and the typical NoSQL systems in various ways. The unique selling point is the build-in notion of time which allows to use the same queries over the current database as over all previous versions of it. With the reads untangled from the writes, the line between of OLTP and OLAP dissolves. With the separation of the architectural parts forming a database system, Datomic offers at least some opportunities for scaling out.

The future has to show if the ideas behind Datomic lead to a whole new class of database systems. Currently there is only one proprietary implementation available which is driven by the inventor of Clojure Rich Hickey, and two co-founders of Relevance, Inc. Stuart Halloway and Justin Gehtland.

References

[CO96] Chris Okasaki, "Purely functional data structures", 1996

[CL] http://clojure.org

[DA] http://www.datomic.com

[LG02] Nancy Lynch and Seth Gilbert, "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services", ACM SIGACT News, Volume 33 Issue 2 (2002), pg. 51-59

[FK01] Amos Fiat and Haim Kaplan, "Making Data Structures Confluently Persistent", 2001

[DST94] J. Driscoll, D. Sleator, R. Tarjan, "Fully persistent lists with catenation", Journal of the ACM, 41(5):943-959, 1994

[1] https://groups.google.com/d/msg/datomic/jYjWvut3v3E/GZeod3hYaRQJ