Eventually Consistent Transactions

Sebastian Burckhardt[1], Daan Leijen[1], Manuel Fähndrich[1], and Mooly Sagiv[2]

[1] Microsoft Research
[2] Tel-Aviv University

Abstract. When distributed clients query or update shared data, eventual consistency can provide better availability than strong consistency models. However, programming and implementing such systems can be difficult unless we establish a reasonable consistency model, i.e. some minimal guarantees that programmers can understand and systems can provide effectively.

To this end, we propose a novel consistency model based on *eventually consistent transactions*. Unlike serializable transactions, eventually consistent transactions are ordered by two order relations (visibility and arbitration) rather than a single order relation. To demonstrate that eventually consistent transactions can be effectively implemented, we establish a handful of simple operational rules for managing replicas, versions and updates, based on graphs called *revision diagrams*. We prove that these rules are sufficient to guarantee correct implementation of eventually consistent transactions. Finally, we present two operational models (single server and server pool) of systems that provide eventually consistent transactions.

1 Introduction

Eventual Consistency [17] is a well-known workaround to the fundamental problem of providing CAP [9] (consistency, availability, and partition tolerance) to clients that perform queries and updates against shared data in a distributed system. It weakens traditional consistency guarantees (such as linearizability) in order to allow clients to perform updates against any replica, at any time. Eventually consistent systems guarantee that all updates are eventually delivered to all replicas, and that they are applied in a consistent order.

Eventual consistency is popular with system builders. One reason is that it allows temporarily disconnected replicas to remain fully available to clients. This is particularly useful for implementing clients on mobile devices [20]. Another reason is that it does not require updates to be immediately performed on all server replicas, thus improving scalability. In more theoretical terms, the benefit of eventual consistency can be understood as its ability to *delay* consensus [16].

However, eventual consistency is a weak consistency model that breaks with traditional approaches (e.g. serializable operations) and thus requires developers to be more careful. The essential problem is that updates are not immediately applied globally, thus the conditions under which they are applied are subject to change, which can easily break data invariants. Many eventually consistent systems address this issue by providing higher-level data types to programmers. Still, the semantic details often remain sketchy. Experience has shown that ad-hoc approaches to the semantics and implementation of such systems can lead to surprising behaviors (e.g. a shopping cart where deleted items reappear [7]). To take eventual consistency to its full potential, we need answers to the following questions:

- How can we provide consistency guarantees that are as strong as possible without forsaking lazy consensus?
- How can we effectively understand and implement systems that provide those guarantees?

In this paper, we propose a two-pronged solution that addresses both questions, based on (1) a notion of transactions for eventual consistency, and (2) a general implementation technique based on revision diagrams.

Eventually consistent transactions differ significantly from traditional transactions, as they are not serializable. Nevertheless, they uphold traditional atomicity and isolation guarantees. Even better, they exhibit some strong properties that simplify the life of programmers and are not typically offered by traditional transactions: (1) transactions cannot fail and never roll back, and (2) all code, even long-running tasks, can run inside transactions without compromising performance.

We first present an abstract, concise specification of eventually consistent transactions. This formalization uses mathematical techniques (sets of events, partial orders, and equivalence relations) that are commonly used in research on relaxed memory models and transactional memory. Our definition provides immediate insight on how eventual consistency is related to strong consistency: the only difference is that eventual consistency uses two separate order relations (visibility order and arbitration order) rather than a single order over transactions.

We then proceed to describe a more concrete and operational implementation technique based on *revision diagrams* [6]. Revision diagrams provide implementors with a simple set of rules for managing updates and replicas. Revision diagrams make the fork and join of versions explicit, which determines the visibility and arbitration of transactions. We prove a theorem that guarantees that any system following the revision diagram rules provides eventually consistent transactions according to the abstract definition. We also illustrate the use of revision diagrams by presenting two simple system models (one using a single server, and one using a server pool).

Overall, we make the following contributions:

- We introduce a notion of *eventually consistent transactions* and give a concise and abstract definition.
- We present a systematic approach for building systems that support such transactions, based on *revision diagrams*. We present a precise, operational definition of revision diagrams.
- We prove a theorem stating that the revision diagram rules are sufficient to guarantee eventual consistency. The proof is nontrivial as it depends on deep structural properties of revision diagrams.
- We illustrate the use of revision diagrams by presenting two operational system models, using a single server and a server pool, respectively.

2 Formulation

To get started, we need to establish some precise terminology. Perhaps the very first question is: what is a database? At a high abstraction level, databases are no different than abstract data types, which are semantically defined by the operations they support to update them and retrieve data. Taking cues from common definitions of abstract data types, we define:

Definition 1. *A* query-update *interface is a tuple* (Q, V, U) *where* Q *is an abstract set of query operations,* V *is an abstract set of values returned by queries, and* U *is an abstract set of update operations.*

Note that the sets of queries, query results, and updates are not required to be finite (and usually are not). Query-update interfaces can apply in various scenarios, where they may describe abstract data types, relational databases, or simple random-access memory, for example. For databases, queries are typically defined recursively by a query language.

Example 1. Consider random-access memory that supports loads and stores of bytes in a 64-bit address space $A = \{a \in \mathbb{N} \mid 0 < a \leq 2^{64}\}$. For that example we define $Q = \{\mathit{load}(a) \mid a \in A\}$, $V = \{v \in \mathbb{N} \mid 0 < v \leq 2^8\}$ and $U = \{\mathit{store}(a, v) \mid a \in A \text{ and } v \in V\}$.

This example is excellent for illustration purposes (we will revisit it throughout), and it provides an explicit connection between our results and previous work on relaxed memory models and transactional memory. Of course, most databases also fit in this abstract interface where the queries are SQL queries and the update operations are SQL updates like insertion and deletion.

So far, our interfaces have no inherent meaning. The most direct way to define the semantics of queries and updates is to relate them to some notion of state:

Definition 2. *A* query-update automaton *(QUA) for the interface* (Q, V, U) *is a tuple* (S, s_0) *where* S *is a set of states with (1) an initial state* $s_0 \in S$*, (2) an interpretation* $q^{\#}$ *of each query* $q \in Q$ *as a function* $S \to V$*, and (3) an interpretation* $u^{\#}$ *of each update operation* $u \in U$ *as a a function* $S \to S$*.*

Example 2. The random-access memory interface described in Example 1 above can be represented by a QUA (S, s_0) where S is the set of total functions $A \to V$, and where s_0 is the constant function that maps all locations to zero, and where $\mathit{load}(a)^{\#}(s) = s(a)$ and $\mathit{store}(a, v)^{\#}(s) = s[a \mapsto v]$.

QUAs can naturally support abstract data types (e.g. collections, or even entire documents) that offer higher-level operations (queries and updates) beyond just loads and stores. Such data types are often important when programming against a weak consistency model [18], since they can ensure that the data representation remains intact when handling concurrent and potentially conflicting updates.

The following two characteristics of QUAs are important to understand how they relate to other definitions of abstract data types:

- There is a strict separation between query and update operations: it is not possible for an operation to both update the data *and* return information to the caller.
- All updates are total functions. It is thus not possible for an update to 'fail'; however, it is of course possible to define updates to have no effect in the case some precondition is not satisfied.

For instance, in our formalization, we would not allow a classic stack abstract data type with a pop operation for two reasons, (1) pop both removes the top element of the stack and returns it, so it is neither an update nor a query, and (2) pop is not total, i.e. it can not be applied to the empty stack.

This restriction is crucial to enable eventual consistency, where the sequencing and application of updates may be delayed, and updates may thus be applied to a different state than the one in which they were originally issued by the program.

2.1 Clients and Transactions

Things become more interesting and challenging once we consider a distributed system. We call the participants of our system *clients*. Clients typically reside on physically distinct devices, but are not required to do so. When clients in a distributed system issue queries and updates against some shared QUA, we need to define what consistency programmers can expect. This consistency model should also address the semantics of *transactions*, which provide clients with the ability to perform several updates as an atomic "bundle".

We formally represent this scenario by defining a set C of clients. Each client, at its own speed, issues a sequence of transactions. Supposedly, each client runs some form of program (the details of which we leave unspecified for simplicity and generality). This program determines when to begin and end a transaction, and what operations to perform in each transaction, which may depend on various factors, such as the results returned by queries, or external factors such as user inputs.

For uniformness, we require that all operations are part of a transaction. This assumption comes at no loss of generality: a device that does not care about transactions can simply issue each operation in its own transaction.

Since all operations are inside transactions, we need not distinguish between the end of a transaction and the beginning of a transaction. Formally, we can thus represent the activities on a device as a stream of operations (queries or updates) interrupted by special yield operations that mark the transaction boundary.[1]

We can thus fully describe the interaction between programs executing on the clients and the database by the following three types of operations:

1. Updates $u \in U$ issued by the program,
2. Pairs (q, v) representing a query $q \in Q$ issued by the program, together with a response $v \in V$ by the database system,
3. The yield operations issued by the program.

[1] We call this operation yield() since it is semantically similar to a yield we may encounter on a uniprocessor performing cooperative multittasking: such a yield marks locations where other threads may read and modify the current state of the data, while at all other locations, only the current thread may read or modify the state.

Definition 3. *A* history *H for a set C of clients and a query-update interface (Q, V, U) is a map H which maps each client $c \in C$ to a finite or infinite sequence $H(c)$ of operations from the alphabet $\Sigma = U \cup (Q \times V) \cup \{\mathsf{yield}\}$.*

Note that our history does not a priori include a global ordering of events, since such an order is not always meaningful when working with relaxed consistency models. Rather, the existence of certain orderings, subject to certain conditions, is what determines whether a history satisfies a consistency model or not.

Notation and Terminology. To reason about a history H, it is helpful to introduce the following auxiliary terminology. We let E_H be the set of all *events* in H, by which we mean all occurrences of operations in $\Sigma \setminus \{\mathsf{yield}\}$ in the sequences $H(c)$ (we consider yield to be just a marker within the operation sequence, but not an event).

For a client c, we call a maximal nonempty contiguous subsequence of events in $H(c)$ that does not contain yield a *transaction* of c. We call a transaction *committed* if it is succeeded by a yield operation, and uncommitted otherwise. We let T_H be the set of all transactions of all clients, and *committed*$(T_H) \subseteq T_H$ the subset of all committed transactions. For an event e, we let $\mathit{trans}(e) \in T_H$ be the transaction that contains e. Moreover, we let *committed*$(E_H) \subseteq E_H$ be the subset of events that are contained in committed transactions. We conclude by giving definitions related to ordering events and transactions:

- *Program order*. For a given history H, we define a partial order $<_p$ over events in H such that $e <_p e'$ iff e appears before e' in some sequence $H(c)$.
- *Apply in order*. For a history H, for a state $s \in S$, for a subset of events $E' \subset E_H$, and for a total order $<$ over the events in E', we let $\mathit{apply}(E', <, s)$ be the state obtained by applying all updates appearing in E' to the state s, in the order specified by $<$.
- *Factoring*. We define an equivalence relation $\sim_t$ (same-transaction) over events such that $e \sim_t e'$ iff $\mathit{trans}(e) = \mathit{trans}(e')$. For any partial order $\prec$ over events, we say that $\prec$ *factors* over $\sim_t$ iff for any events x and y from different transactions, $x \prec y$ implies $x' \prec y'$ for any x, y such that $x \sim_t x'$ and $y \sim_t y'$. This is an important property to have for any ordering $\prec$, since if $\prec$ factors over $\sim_t$, it induces a corresponding partial order on the transactions.

2.2 Sequential Consistency

Sequential consistency posits that the observed behavior must be consistent with an interleaving of the transactions by the various devices. We formalize this interleaving as a partial order over events (rather than a total order as more commonly used) since some events are not instantly ordered by the system; for example, the relative order of operations in uncommitted transactions may not be fully determined yet.

Definition 4. *A history H is* sequentially consistent *if there exists a partial order $<$ over the events in E_H that satisfies the following conditions for all events $e_1, e_2, e \in E_H$:*

- *(compatible with program order) if* $e_1 <_p e_2$ *then* $e_1 < e_2$
- *(total order on past events) if* $e_1 < e$ *and* $e_2 < e$ *then either* $e_1 < e_2$ *or* $e_2 < e_1$.
- *(consistent query results) for all* $(q, v) \in E_H$, $v = q^{\#}(apply(\{e \in (H) \mid e < q\}, <, s_0))$. *This simply says that a query returns the state as it results from applying all past updates to the initial state.*
- *(atomicity)* $<$ *factors over* $\sim_t$.
- *(isolation) if* $e_1 \notin committed(E_H)$ *and* $e_1 < e_2$, *then* $e_1 <_p e_2$. *That is, events in uncommitted transactions precede only events on the same client.*
- *(eventual delivery) for all committed transactions* $t \in committed(T_H)$, *there exist only finitely many transactions* $t' \in T_H$ *such that* $t \not< t'$.

Sequential consistency fundamentally limits availability in the presence of network partitions. The reason is that any query issued by some transaction t *must* see the effect of all updates that occur in transactions that are globally ordered before t, even if on a remote device. Thus we cannot conclusively commit transactions in the presence of network partitions.

2.3 Eventual Consistency

Eventual consistency relaxes sequential consistency by allowing queries in a transaction t to see only a subset of all transactions that are globally ordered before t. It does so by distinguishing between a visibility order (a partial order that defines what updates are visible to a query), and an arbitration order (a partial order that determines the relative order of updates).

Definition 5. *A history* H *is* eventually consistent *if there exist two partial orders* $<_v$ *(the visibility order) and* $<_a$ *(the arbitration order) over events in* H*, such that the following conditions are satisfied for all events* $e_1, e_2, e \in E_H$*:*

- *(arbitration extends visibility) if* $e_1 <_v e_2$ *then* $e_1 <_a e_2$.
- *(total order on past events) if* $e_1 <_v e$ *and* $e_2 <_v e$, *then either* $e_1 <_a e_2$ *or* $e_2 <_a e_1$.
- *(compatible with program order) if* $e_1 <_p e_2$ *then* $e_1 <_v e_2$.
- *(consistent query results) for all* $(q, v) \in E_H$, $v = q^{\#}(apply(\{e \in H) \mid e <_v q\}, <_a, s_0))$. *This says that a query returns the state as it results from applying all preceding* visible *updates (as determined by the visibility order) to the initial state, in the order given by the arbitration order.*
- *(atomicity) Both* $<_v$ *and* $<_a$ *factor over* $\sim_t$.
- *(isolation) if* $e_1 \notin committed(E_H)$ *and* $e_1 <_v e_2$, *then* $e_1 <_p e_2$. *That is, events in uncommitted transactions are visible only to later events by the same client.*
- *(eventual delivery) for all committed transactions* $t \in committed(T_H)$, *there exist only finitely many transactions* $t' \in T_H$ *such that* $t \not<_v t'$.

The reason why eventual consistency can tolerate temporary network partitions is that the arbitration order can be constructed incrementally, i.e. may remain only partially determined for some time after a transaction commits. This allows conflicting updates to be committed even in the presence of network partitions.

Note that eventual consistency is a weaker consistency model than sequential consistency. We can prove this statement as follows.

Lemma 1. *A sequentially consistent history is eventually consistent.*

Proof. Given a history H that is sequentially consistent, we know there exists a partial order $<$ satisfying all conditions. Now define $<_v = <_a = <$; then all conditions for eventual consistency follow easily.

2.4 Eventual Consistency in Related Work

Eventual consistency across the literature uses a variety of techniques to propagate updates (e.g. general causally-ordered broadcast [18,19], or pairwise anti-entropy [15]). All of these techniques are particular implementations that specialize our general definition of visibility as a partial order. As for the arbitration order, we found that two main approaches prevail. The most common one is to use (logical or actual) *timestamps*: Timestamps provide a simple way to arbitrate events. Another approach (sometimes combined with timestamps) is to make updates *commutative*, which makes arbitration unnecessary (i.e. we can pick an arbitrary serialization of the visibility order to satisfy the conditions in Def. 5).

We show in the next section (Section 3) how to arbitrate updates without using timestamps or requiring commutativity, a feature that sets our work apart. We prefer to not use timestamps because they exhibit the *write stabilization* problem [20], i.e. the inability to finalize the effect of updates while older updates may still linger in disconnected network partitions. Consider, for example, a mobile user called Robinson performing an important update, but getting stranded on a disconnected island before transmitting it. When Robinson reconnects after years of exile, Robinson's update is older than (and may thus alter the effect of) all the updates committed by other users in the meantime. So either (1) none of these updates can stabilize until Robinson returns, or (2) after some timeout we give up on Robinson and discard his update. Clearly, neither of these solutions is satisfactory. A better solution is to abandon time stamps and instead use an arbitration order that simply orders Robinson's update *after* all the other updates. In fact, this is the outcome we achieve when using revision diagrams, as explained in Section 3.

3 Revision Consistency

Our definition of eventual consistency (Def. 5) is concise and general. By itself, it is however not very constructive, insofar that it does not give practical guidelines as to how a system can efficiently and correctly construct the necessary ordering (visibility and arbitration). We now proceed to describe a more specific implementation technique for eventually consistent systems, based on the notion of *revision diagrams* introduced in [6].

Revision diagrams show an extended history not only of the queries, updates, and transactions by each client, but also of the forking and joining of *revisions*, which are logical replicas of the state (Fig. 1). A client works with one revision at a time, and can perform operations (queries and updates) on it. Since different clients work with different revisions, clients can perform both queries and updates concurrently and in isolation (i.e. without creating race conditions). Reconciliation happens during *join* operations. When a revision *joins* another revision, it replays all the updates performed in the joined revision at the join point.[2] After a revision is joined, no more operations can be performed on it (i.e. clients may need to fork new revisions to keep enough revisions available).

3.1 Revision Diagrams

Revision diagrams are directed graphs constructed from three types of edges (successor, fork, and join edges, or s-, f- and j-edges for short), and five types of vertices (start, fork, join, update, and query vertices). A start vertex represents the beginning of a revision, s-edges represent successors within a revision, and fork/join edges represent the forking and joining of revisions.

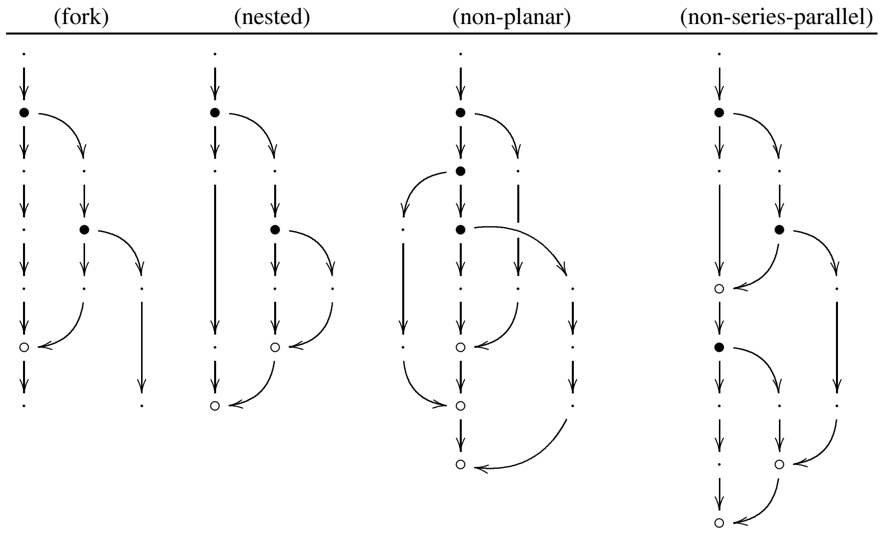

Fig. 1. Examples of Valid Revision Diagrams

We pictorially represent revision diagrams using the following conventions

- Use · for start, query, and update vertices
- Use • and ∘ for fork and join vertices, respectively
- Use vertical down-arrows for s-edges
- Use horizontal-to-vertical curved arrows for f-edges
- Use vertical-to-horizontal curved arrows for j-edges

A vertex x has a s-path (i.e. a path contanining only s-edges) to vertex y if and only if they are part of the same revision. Since all s-edges are vertical in our pictures,

[2] This replay operation is *conceptual*. Rather than replaying a potentially unbounded log, actual implementations can often use much more space- and time-efficient merge functions, as explained in Section 4.

vertices belonging to the same revision are always aligned vertically. For any vertex x we let $S(x)$ be the start vertex of the revision that x belongs to. For any vertex x whose start vertex $S(x)$ is not the root, we define $F(x)$ to be the fork vertex such that $F(x) \xrightarrow{f} S(x)$ (i.e. the fork vertex that started the revision x belongs to). We call a vertex with no outgoing s- or j-edges a *terminal*; terminals are the last operation in a revision that can still perform operations (has not been joined yet), and thus represent potential extension points of the graph.

We now give a formal, constructive definition for revision diagrams.

Definition 6. *A* revision diagram *is a directed graph constructed by applying a (possibly empty or infinite) sequence of the following construction steps (see Fig 2(a)) to a single initial start vertex (called the root):*

Query. *Choose some terminal t, create a new query vertex x, and add an edge $t \xrightarrow{s} x$.*
Update. *Choose some terminal t, create a new update vertex x, and add an edge $t \xrightarrow{s} x$.*
Fork. *Choose some terminal t, create a new fork vertex x and a new start vertex y, and add edges $t \xrightarrow{s} x$ and $x \xrightarrow{f} y$.*
Join. *Choose two terminals t, t' satisfying the **join condition** $F(t') \rightarrow^* t$, then create a new join vertex x and add edges $t \xrightarrow{s} x$ and $t' \xrightarrow{j} x$.*

The join condition expresses that the terminal t (the "joiner") must be reachable from the *fork* vertex that started the revision that contains t' (the "joinee"). This condition makes revision diagrams more restricted than general task graphs. See Fig 2(b) for some examples of invalid diagrams where the join condition does not hold at construction of the join nodes.

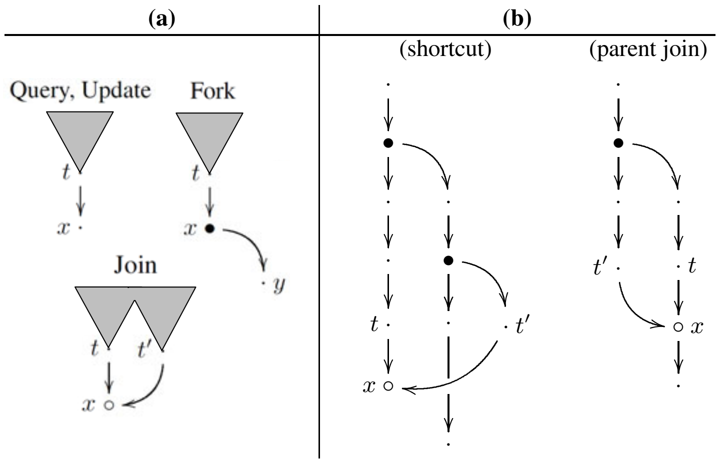

Fig. 2. **(a)** (left) Visualization of the construction rules for revision diagrams in Def. 6. **(b)** (right) Examples of *invalid* revision diagrams. Both diagrams are not possible since they violate the join property at the creation of the join node x. Note that in the right diagram, $F(t')$ is undefined on the main revision and therefore $F(t') \rightarrow^* t$ does not hold.

The join condition has some important, not immediately obvious consequences. For example, it implies that revision diagrams are always semilattices (for a proof of this nontrivial fact see [6]). Also, it ensures some diagram properties (Lemmas 2 and 3) that we need to prove our main result (Thm. 1). Futhermore, it still allows more general graphs than strict series-parallel graphs [21], which allow only the recursive serial and parallel composition of tasks (and are also called fork-join concurrency in some contexts, which is potentially misleading). For instance, the right-most revision diagram in Fig. 1 is not a series-parallel graph but it is a valid revision diagram. While series-parallel graphs are easier to work with than revision diagrams, they are not flexible enough for our purpose, since they would enforce too much synchronization between participants.

Also, note that fork and the join are fundamentally asymmetric: the revision that initiates the fork (the "forker") continues to exist after the fork, but also starts a new revision (the "forkee"), and similarly, the revision that initiates the join (the "joiner") can continue to perform operations after the join, but ends the joined revision (the "joinee").

3.2 Graph Properties

We now examine some properties of the revision diagrams, for better visualization, and because we need some technical properties in our later proofs. Most statements are easily proved by induction over the construction rules in Def. 6; if not, we mention how to prove them.

Revision diagrams are connected, and all vertices are reachable from the root vertex. There can be multiple paths from the root to a given vertex, but exactly one of those is free of j-edges.

Definition 7. *For any vertex v in a revision diagram, let the* root-path *of v be the unique path from the root to v that does not contain j-edges.*

The join condition does not make revision diagrams necessarily planar, i.e. when drawing revision diagrams, it is not always possible to avoid crossing lines (see the third diagram in Fig. 1 for an example). However, it is always possible to choose horizontal coordinates for the vertices such that (1) vertices in the same revisions are vertically aligned, and (2) revisions are horizontally arranged such that forkers are left of forkees, and (3) joiners are left of joinees. The existence of such an order is not immediately obvious; for example, such a layout is not possible for the incorrect revision diagram at the right in Fig. 2(b). The following lemma formalizes the claims (1,2,3) above (where the preorder $\leq_l$ corresponds to a relation on vertices that compares their horizontal coordinates):

Lemma 2. *[Layout Preorder] In any revision diagram, there exists a preorder $\leq_l$ on vertices*[3] *such that*

$$S(x) = S(y) \quad \Leftrightarrow \quad (x \leq_l y) \wedge (y \leq_l x) \tag{1}$$

$$x \xrightarrow{f} y \quad \Rightarrow \quad x \leq_l y \tag{2}$$

$$x \xrightarrow{j} y \quad \Rightarrow \quad y \leq_l x \tag{3}$$

[3] A *preorder* is a reflexive transitive binary relation. Unlike partial orders, preorders are not necessarily antisymmetric, i.e. they may contain cycles.

We include the proof in the full version [4]. For proving our main result later on, we need to establish another basic fact about revision diagrams. We call a path *direct* if all of its f-edges (if any) appear after all of its j-edges (if any). The following lemma (which appears as a theorem in [6], and for which we include a proof in [4] as well) shows that we can always choose direct paths:

Lemma 3 (Direct Paths.). *Let x, y be vertices in a revision diagram. If $x \rightarrow^* y$, there exists a direct path from x to y.*

3.3 Query and Update Semantics

We now proceed to explain how to determine the results of a query in a revision diagram. The basic idea is to (1) return a result that is consistent with applying all the updates along the root path, and (2) if there are join vertices along that path, they summarize the effect of *all* updates by the joined revision.

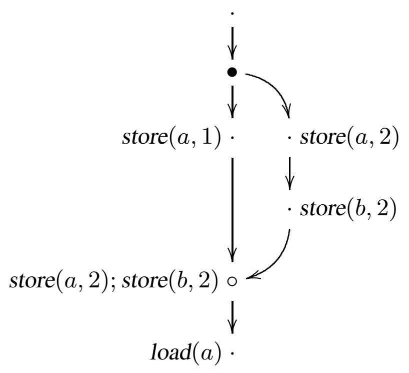

Fig. 3. A labeled revision diagram. The *path-result* of the bottom vertex is now the query applied to its root-path: $\mathit{load}(a)^{\#}(\mathit{store}(b,2)^{\#}(\mathit{store}(a,2)^{\#}(\mathit{store}(a,1)^{\#}(s_0)))) = 2$.

For example, consider the diagram in Fig. 3. This is an example of a revision diagram labeled with the operations of the random access memory example described in Example 2. The join vertex is labeled with the composition of all update operations of the joinee. The *path-result* of the final query node $\mathit{load}(a)$ can now be evaluated by applying to the composition of all update operations along the root-path: $\mathit{load}(a)^{\#}(\mathit{store}(b,2)^{\#}(\mathit{store}(a,2)^{\#}(\mathit{store}(a,1)^{\#}(s_0)))) = 2$.

We can define this more formally. To reduce the verbosity of our definitions, we assume a fixed query-update interface (Q, V, U) and QUA (S, s_0) for the rest of this section.

Definition 8. *For any vertex x, we let the* effect *of x be a function $x^\circ : S \rightarrow S$ defined inductively as follows:*

- *If x is a start, fork, or query vertex, the effect is a no-op, i.e. $x^\circ(s) = s$.*
- *If x is an update vertex for the update operation u, then the effect is that update, i.e. $x^\circ(s) = u^{\#}(s)$.*

– *If x is a join vertex, then the effect is the composition of all effects in the joined revision, i.e. if $y_1, \ldots, y_n$ is the sequence of vertices in the joined revision (i.e. y_1 is a start vertex, $y_i \xrightarrow{s} y_{i+1}$ for all $1 \leq i < n$, and $y_n \xrightarrow{j} x$), then $x^\circ(s) = y_n^\circ(y_{n-1}^\circ(\ldots y_1^\circ(s)))$.*

We can then define the expected query result as follows.

Definition 9. *Let x be a query vertex with query q, and let $(y_1, \ldots, y_n, x)$ be the root path of x. Then define the* path-*result of x as $q^\#(y_n^\circ(y_{n-1}^\circ(\ldots y_1^\circ(s_0))))$.*

3.4 Revision Diagrams and Histories

We can naturally relate histories to revision diagrams by associating each query event $(q, v) \in E_H$ with a query vertex, and each update event $u \in E_H$ with a update vertex. The intention is to validate the query results in the history using the path results, and to keep transactions atomic and isolated by ensuring that their events form contiguous sequences within a revision.

Definition 10. *We call a revision diagram a* witness *for the history H if it satisfies the following conditions:*

1. *For all query events (q, v) in E_H, the value v matches the path-result of the query vertex.*
2. *If x, y are two successive non-*yield *operations in $H(c)$ for some c, then they must be connected by a s-edge.*
3. *If x is the last event of $H(c)$ for some c and not a* yield*, then it must be a terminal.*
4. *If x, y are two operations preceding and succeeding some* yield *in $H(c)$ for some c, then there must exist a path from x to y. In other words, the beginning of a transaction must be reachable from the end of the previous transaction.*

We call a history H revision-consistent *if there exists a witness revision diagram.*

To ensure eventual delivery of updates, we need to somehow make sure there are enough forks and joins. To formulate a liveness condition on infinite histories, we define "neglected vertices" as follows:

Definition 11. *We call a vertex x in a revision diagram* neglected *if there exists an infinite number of vertices y such that there is no path from x to y.*

We are now ready to state and prove our main result.

Theorem 1. *Let H be a history. If there exists a witness diagram for H such that no committed events are neglected, then H is eventually consistent.*

Note that this theorem gives us a solid basis for implementing eventually consistent transactions: an implementation can be based on dynamically constructing a witness revision diagram and as a consequence guarantee eventual consistent transactions. Moreover, as we will see in Section 4, implementations do not need to actually construct such witness diagrams at runtime but can rely on efficient state-based implementations.

The proof of our Theorem (in Section 3.5 below) constructs partial orders $<_v, <_a$ from the revision diagram by (1) specifying $x <_v y$ iff there is a path from x to y in the revision diagram, and (1) specifying $<_a$ to order all events in a joined revision to occur in between the joiner terminal and the join vertex. Note that the converse of Thm. 1 is not true, not even if restricted to finite histories (we include a finite counterexample in the full version [4]). Also Note that the most difficult part of the proof is the safety, not the liveness, since the proof that $<_a$ is a partial order extending $<_v$ depends on the join condition in a nontrivial way.

3.5 Proof of Thm 1

We devote the rest of this section to this proof, which requires some deeper insight into structural properties of revision diagrams. First, however, we need some definitions, notations, and lemmas.

A revision diagrams is a connected graph. However, if we remove all f-edges from the picture, it may decompose into several components. We define a *join-component* to be a maximal component connected by s and j edges only. We say $x \sim_j y$ if they are in the same join component, and let $J(x) = \{y \mid x \sim_j y\}$. It is easy to see that each join-component contains exactly one terminal. For a vertex x, we let $T(x)$ be the terminal of $J(x)$ (note that $T(x)$ is the unique terminal reachable from x by a path containing j and s edges only).

Definition 12. *Define the binary relation $\rightarrow_a$ on vertices by adding the following edges during the construction of a revision diagram as in Def. 6:*

- ***(Query, Update, Fork)*** *for all $y \in J(t)$, add $y \rightarrow_a x$*
- ***(Join)*** *for all $y \in J(t)$ and $y' \in J(t')$, add edges $y \rightarrow_a x$, $y' \rightarrow_a x$, and $y \rightarrow_a y'$.*

Lemma 4. *For any revision diagram, $\rightarrow_a$ as defined above is a partial order over all vertices in the diagram satisfying (1) when restricted to any one join-component, $\rightarrow_a$ is a total order (2) $\rightarrow_a$ does not cross join-components.*

Lemma 5. *For vertices x, y in a revision diagram and a preorder $\leq_l$ as guaranteed by Lemma 2, $x \rightarrow^* y$ implies $T(x) \leq_l T(y)$.*

We include proofs for both lemmas in [4]. The first one is a simple induction, the second one is a bit more intricate and uses the path properties guaranteed by Lemma 3 and the layout preorder guaranteed by Lemma 2.

We are now ready to prove Theorem 1. Given a history H and a witness revision diagram, define two binary relations

$$<_v = \rightarrow^* \quad \text{and} \quad <_a = (<_v \cup \rightarrow_a)^*.$$

By Lemma 6 below, $<_a$ and $<_v$ are partial orders. We can then prove the remaining claims as follows:

- (arbitration extends visibility) By Lemma 6 below.

- (total order on past events) if $e_1 <_v e$ and $e_2 <_v e$, then by Lemma 3 there exist direct paths for $e_1 \rightarrow^* e$ and for $e_2 \rightarrow^* e$. If either path is a prefix of the other, e_1 and e_2 are ordered by $<_v$ and thus by $<_a$. If not, they must combine in a join vertex, implying that $e_1 \sim_j e_2$, which implies (by Lemma 4) that they are ordered by $<_a$.
- (compatible with program order) By conditions 2 and 4 of Def. 10.
- (consistent query results) We can show inductively (over Def. 6) that for any vertex x, the combined effect of the vertices on the root path (as in Def. 8) to x is equal to the combined effect of all updates $\{x' \mid x' <_v x\}$ ordered by $<_a$. This is trivial for all but the join case. In the join case, Def. 12 orders all all updates in the joinee after updates in the joiner which is consistent with interpreting them as an effect of the join vertex.
- (atomicity) By condition 2 we know there can be no intervening forks or joins. This implies that both $\rightarrow$ and $<_a$ factor over $\sim_t$.
- (isolation) By condition 3.
- (eventual delivery) Assume the condition is violated. Then there exists a committed transaction $t \in$ *committed*(T_H) and an infinite number of transactions $t_1, t_2, \ldots$ such that for all $i, t \not<_v t_i$. Since transactions can not be empty, we can pick vertices $x \in t$ and $x_i \in t_i$, with $x \not<_v x_i$ for all i. But that implies that x is neglected, contradicting the condition in the theorem.

The only thing left to prove is the lemma below, which arguably contains the most interesting part of the proof. In particular, it shows how consequences of the join condition (specifically, Lemmas 2 and 5) are used in the construction of an arbitration order $<_a$ that satisfies $<_v \subseteq <_a$ as required for eventual consistency.

Lemma 6. *Given some revision diagram, define binary relations $<_v = \rightarrow^*$ and $<_a = (<_v \cup \rightarrow_a)^*$. Then both $<_v$ and $<_a$ are partial orders, and $<_v \subseteq <_a$.*

Proof. Clearly, $<_v$ is a partial order (since revision diagrams are acyclic) and $<_v \subseteq <_a$. The interesting part is to show that $<_a$ is antisymmetric (i.e. $x <_a y$ and $y <_a x$ implies $x = y$). We prove this by showing that $(\rightarrow_a \cup \rightarrow)$ is acyclic. Consider some minimal cycle. Since $\rightarrow_a$ is transitive, and both $\rightarrow_a$ and $\rightarrow$ are acyclic on their own, it must be of the following form (where $n \geq 1$):

$$x_1 \rightarrow^* y_1 \rightarrow_a x_2 \rightarrow^* y_2 \rightarrow_a \ldots \rightarrow_a x_n \rightarrow^* y_n \rightarrow_a x_1$$

By Lemma 4 this implies

$$x_1 \rightarrow^* y_1 \sim_j x_2 \rightarrow^* y_2 \sim_j \ldots \rightarrow_a x_n \rightarrow^* y_n \sim_j x_1$$

using the preorder guaranteed by Lemma 2 and Lemma 5, we get

$$T(x_1) \leq_l T(y_1) = T(x_2) \leq_l T(y_2) \ldots T(x_n) \leq_l T(y_n) = T(x_1)$$

But by Lemma 2 such an $\leq_l$-cycle implies that all vertices are in the same revision which is a contradiction.

4 System Implementation

Revision diagrams can help to develop efficient implementations since they provide a solid abstraction that decouples the consistency model from actual implementation choices. In this section, we describe some implementation techniques that are likely to be useful for that purpose. We present three sketches of client-server systems that implement eventual consistency.

It is usually not necessary for implementations to store the actual revision diagram. Rather, we found it highly convenient to work with state representations that can directly provide fork and join operations.

Definition 13. *A* fork-join QUA *(FJ-QUA) for a query-update interface (Q, V, U) is a tuple (Σ, σ_0, f, j) where (1) (Σ, σ_0) is a QUA over (Q, V, U), (2) $f : \Sigma \to \Sigma \times \Sigma$, and (3) $j : \Sigma \times \Sigma \to \Sigma$.*

If we have a fork-join QUA, we can simply associate a Σ-state with each revision, and then perform all queries and updates locally on that state, without communicating with other revisions. The join function of the FJ-QUA, if implemented correctly, guarantees that all updates are applied at the join time. We can state this more formally as follows.

Definition 14. *For a FJ-QUA (Σ, σ_0, f, j) and a revision diagram over the same interface (Q, V, U), define the state $\sigma(x)$ of each vertex x inductively by setting $\sigma(r) = \sigma_0$ for the initial vertex r, and (for the construction rules as they appear in Def. 6)*

- *(Query) Let $\sigma(x) = \sigma(t)$*
- *(Update) Let $\sigma(x) = u^{\#}(\sigma(t))$*
- *(Fork) Let $(\sigma(x), \sigma(y)) = f(\sigma(t))$*
- *(Join) Let $\sigma(x) = j(\sigma(t), \sigma(t'))$*

Definition 15. *A FJ-QUA (Σ, σ_0, f, j)* implements *the QUA (S, s_0) over the same interface if and only if for all revision diagrams, for all vertices x, the locally computed state $\sigma(x)$ (as in Def. 14) matches the path result (as in Def. 9).*

Example 3. Consider the QUA representing random access memory as defined in Example 2. We can implement this QUA using an FJ-QUA that maintains a "write-set" as follows:

$$\begin{aligned}
\Sigma &= S \times \mathcal{P}(A) \\
\sigma_0 &= (s_0, \emptyset) \\
\textit{load}(a)^{\#}(s, W) &= s(a) \\
\textit{store}(a, v)^{\#}(s, W) &= (s[a \mapsto v], W \cup \{a\}) \\
f(s, W) &= ((s, W), (s, \emptyset)) \\
j((s_1, W_1), (s_2, W_2)) &= (s', W_1 \cup W_2) \quad \text{where } s'(a) = \begin{cases} s_1(a) \text{ if } a \notin W_2 \\ s_2(a) \text{ if } a \in W_2 \end{cases}
\end{aligned}$$

The write set (together with the current state) provides sufficient information to conceptually replay all updates during join (since only the last written value matters). Note that the write set gets cleared on forks.

Since we can store a log of updates inside Σ, it is always possible to provide an FJ-QUA for any QUA (we show this construction in detail in the full version [4]). However, more space-effective implementations are often possible for QUAs since logs are typically compressible. We include several finite-state examples of FJ-QUAs in [4] as well.

4.1 System Models

If we have a FJ-QUA, we can implement eventually consistent systems quite easily. We now present two models that demonstrate this principle.

4.2 Single Synchronous Server Model

We first present a model using a single server. We define the set of devices $I = C \cup \{s\}$ where C is the set of clients and s is the single server. We store on each device i a state from the FJ-QUA, that is, we define $R : I \rightharpoonup \Sigma$. To keep the transition rules simple, we use the notation $R[i \mapsto \sigma]$ to denote the map R modified by mapping i to σ, and we let $R(c \mapsto \sigma)$ be a pattern that matches R, c, and σ such that $R(c) = \sigma$. Each client can perform updates and queries while reading and writing only the local state:

$$\textsc{update}(c, u): \quad \frac{\sigma' = u^{\#}(\sigma)}{R(c \mapsto \sigma) \to R[c \mapsto \sigma']} \qquad \textsc{query}(c, q, v): \quad \frac{q^{\#}(\sigma) = v}{R(c \mapsto \sigma) \to R}$$

As for synchronization, all we need is two rules, one to create a new client (forking the server state), and one to perform the yield on the client (joining the client state into the server, then forking a fresh client state from the server):

$$\textsc{spawn}(c): \quad \frac{c \notin dom\, R \qquad f(\sigma) = (\sigma_1, \sigma_2)}{R(s \mapsto \sigma) \to R[s \mapsto \sigma_1][c \mapsto \sigma_2]} \qquad \textsc{yield}(c): \quad \frac{j(\sigma_1, \sigma_2) = \sigma_3 \qquad f(\sigma_3) = (\sigma_4, \sigma_5)}{R(s \mapsto \sigma_1)(c \mapsto \sigma_2) \to R[s \mapsto \sigma_4][c \mapsto \sigma_5]}$$

Thanks to Theorem 1, we can precisely argue why this system is eventually consistent. By induction over the transitions, we can show that each state σ appearing in R corresponds to a terminal in the revision diagram, and each transition rule manipulates those terminals (applying fork, join, update or query) in accordance with the revision diagram construction rules. In particular, the join condition is always satisfied since all forks and joins are performed by the same server revision. Transactions are not interrupted by forks or joins, and no vertices are neglected: each yield creates a path from the freshly committed vertices into the server revision, from where it must be visible to any new clients, and to any client that performs an infinite number of yields.

An interesting observation is that, if the fork does not modify the left component (i.e. for all $\sigma \in \Sigma$, $f(\sigma) = (\sigma, \sigma')$ for some σ'), the server is effectively stateless, in the sense that it does not store any information about the client. This is a highly desirable characteristics for scalability, and in our experience it is well worth to go through some extra length in defining FJ-QUAs that have this property.

4.3 Server Pool Model

The single server model still suffers some drawbacks. For one, clients performing a yield access both server and client state. This means clients block if they have no connection. Also, a single server may not scale to large numbers of clients.

We can fix both of these issues by using a *server pool* rather than a single server, i.e. we let the set of devices be $I = C \cup S$ where S is a set of server identifiers. Using multiple servers not only improves scalability, but it helps with disconnected operation as well: if we keep one server next to each client (e.g. on the same mobile device), we can guarantee that the client does not block on yield. Servers themselves can perform a sync operation (at any convenient time) to exchange state with other servers.

However, we need to keep additional information in each device to ensure that the join condition is maintained. We do so by (1) storing on each client c a pair (σ, n) where σ is the revision state as before, and n is a counter indicating the current transaction, and (2) storing on each server s a triple (σ, J, L) where σ is the revision state as before, J is the set of servers that s may join, and L is a vectorclock (a partial function $(I \rightharpoonup \mathbb{N})$) indicating for each client the latest transaction of c that s may join.

The transitions that involve the client are then as follows:

$$\textsc{update}(c, u): \quad \frac{\sigma' = u^{\#}(\sigma)}{R(c \mapsto (\sigma, n)) \to R[c \mapsto (\sigma', n)]} \qquad \textsc{query}(c, q, v): \quad \frac{q^{\#}(\sigma) = v}{R(c \mapsto (\sigma, L)) \to R}$$

$$\textsc{spawn}(c): \quad \frac{c \notin dom\, R \qquad f(\sigma) = (\sigma_1, \sigma_2) \qquad L' = L[c \mapsto 0]}{R(s \mapsto (\sigma, J, L)) \to R[s \mapsto (\sigma_1, J, L')][c \mapsto (\sigma_2, 0)]}$$

$$\textsc{yield}(s, c): \quad \frac{L(c) = n \qquad L' = L[c \mapsto n+1] \qquad j(\sigma_1, \sigma_2) = \sigma_3 \qquad f(\sigma_3) = (\sigma_4, \sigma_5)}{R(s \mapsto (\sigma_1, J, L))(c \mapsto (\sigma_2, n)) \to R[s \mapsto (\sigma_4, J, L')][c \mapsto (\sigma_5, n+1)]}$$

The servers can perform forks and joins without involving clients. On joins, servers join the state, take the union of the sets J of joinable servers, and merge the vector clocks (defined as taking the pointwise maximum).

$$\textsc{fork}(s_1, s_2): \quad \frac{s_2 \notin dom\, R \qquad f(\sigma) = (\sigma_1, \sigma_2) \qquad J' = J \cup \{s_2\}}{R(s_1 \mapsto (\sigma, J, L)) \to R[s_1 \mapsto (\sigma_1, J', L)][s_2 \mapsto (\sigma_2, J, L)]}$$

$$\textsc{join}(s_1, s_2): \quad \frac{s_2 \in J_1 \qquad \sigma' = j(\sigma_1, \sigma_2) \qquad J' = J_1 \cup J_2 \qquad L' = merge(L_1, L_2)}{R(s_1 \mapsto (\sigma_1, J_1, L_1))(s_2 \mapsto (\sigma_2, J_2, L_2)) \to R[s_1 \mapsto (\sigma', J', L')][s_2 \mapsto \bot]}$$

Again, we can use Theorem 1 to reason that finite executions of this system are eventually consistent (for infinite executions we need additional fairness guarantees as discussed below). Again, all states σ stored in R correspond to terminals in a revision diagram and are manipulated according to the rules. This time, the join condition is satisfied because of the following invariants: (1) if the set J of server s_1 contains s_2, then s_1's terminal is reachable from the fork vertex that forked s_2's revision, and (2) if $L(c) = n$ for server s, and client c's transaction counter is n, then s' terminal is reachable from the fork vertex that forked c's revision.

Since the transition rules do not contain any guarantees that force servers to synchronize with each other, it is possible to construct infinite executions that violate eventual consistency. Actual implementations would thus likely add a mechanism to guarantee that updates eventually reach the main revision, and that clients that perform an infinite sequence of transactions receive versions from the main revision infinitely often.

5 Related Work

For a high-level comparison of our work with various notions of eventual consistency appearing in the literature, see Section 2.4. Briefly stated, our work is set apart by its unique use of *revision diagrams* to determine both arbitration and visibility, rather than separately using a causally consistent partial order for visibility, and timestamps for arbitration.

There is of course a large body of work on transactions. Most academic work considers strong consistency (serializable transactions) only, and is thus not directly applicable to eventual consistency. Nevertheless there are some similarities, to pick a few:

- [10] provides insight on the limitations of serializable transactions, and proposes similar workarounds as used by eventual consistency (timestamps and commutative updates). However, transactions remain tentative during disconnection.
- Snapshot isolation [8] relaxes the consistency model, but transactions can still fail, and can not commit in the presence of network partitions.
- Coarse-grained transactions [11,14] share with our work the use of abstract data types to facilitate concurrent transactions.
- Automatic Mutual Exclusion [1], like our work, uses yield statements to separate transactions.

Previous work on revisions [2,6,3,5] introduces revision diagrams and conflict resolution. In this paper we feature a simpler, more direct definition using graph construction rules. Also, we pursue a different goal (eventually consistent transactions in a distributed system, rather than deterministic parallel programming). In particular, eventually consistent transactions exhibit pervasive nondeterminism caused by factors that are by definition outside the control of the system, such as network partitions. Also, this paper is the first to give a single, simple formalization of merge functions (FJ-QUAS are optimized implementations of QUAs).

Research on *persistent data types* [13] is related to our definition of FJ-QUAs insofar it concerns itself with efficient implementations of data types that permit retrieval and mutations of past versions. However, it does not concern itself with apects related to transactions or distribution.

Prior work on *operational transformations* [19] can be understood as a specialized form of eventual consistency where updates are applied to different replicas in different orders, but are themselves modified in such a way as to guarantee convergence. This specialized formulation can provide highly efficient broadcast-based real-time collaboration, but poses significant implementation challenges [12].

If we consider transactions with single elements only, it is sensible to compare our work with related work on conflict-free replicated data types (CRDTs) [18] and Bayou's weakly consistent replication [20].

- Our definition is strictly more general than CRDTs [18] in the following sense: From any state-based CRDT we can obtain a FJ-QUA by using the same state and initial state, the same query and update functions, a fork function that creates a new replica and then merges the forker state, and a join function that uses the merge. Note that the definition of strong eventual consistency in [18], just like ours, requires that updates can be applied to any state.
- In Bayou [20], and in the Concurrent Revisions work[6], users can specify how to resolve conflicting updates by writing custom merge functions. At first sight, this may appear more general that QUAs. However, by performing a simple automatic transformation of the QUA and the client program, we can support merge functions for conflict resolution purposes. The reason is that QUAs already allow updates to perform any desired total function. We describe this transformation in the full version [4].

6 Conclusion and Future Work

We have proposed *eventually consistent transactions* as a consistency model that (1) generalizes earlier definitions of eventual consistency and (2) shows how to make some strong guarantees (transactions never fail, all code runs in transactions) to compensate for weak consistency. We have shown that revision diagrams provide a convenient way to build correct implementations of eventual consistency, by relying on just a handful of simple rules that are easily visualized using diagrams.

In future work, we would like to extend the study of the programming model, investigate a selection of basic FJ-QUAs, and ways to combine them. Furthermore, we would like to understand whether stronger consistency guarantees are possible for subclasses of eventually consistent transactions, and whether such classes can be automatically recognized or synthesized.

Acknowledgments. We thank Marc Shapiro for introducing us to a principled world of eventual consistency, and for general guidance. We also thank Tom Ball, Sean McDirmid and Benjamin Wood for inspired discussions, helpful examples, and constructive comments.

References

1. Abadi, M., Birrell, A., Harris, T., Isard, M.: Semantics of transactional memory and automatic mutual exclusion. In: Principles of Programming Languages, POPL (2008)
2. Burckhardt, S., Baldassin, A., Leijen, D.: Concurrent programming with revisions and isolation types. In: Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA (2010)
3. Burckhardt, S., Leijen, D., Fähndrich, M.: Roll forward, not back: A case for deterministic conflict resolution. In: Workshop on Determinism and Correctness in Parallel Progr. (2011)
4. Burckhardt, S., Leijen, D., Fähndrich, M., Sagiv, M.: Eventually consistent transactions (full version). Technical Report MSR-TR-2011-117, Microsoft (2011)
5. Burckhardt, S., Leijen, D., Yi, J., Sadowski, C., Ball, T.: Two for the price of one: A model for parallel and incremental computation (distinguished paper award). In: Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA (2011)
6. Burckhardt, S., Leijen, D.: Semantics of Concurrent Revisions. In: Barthe, G. (ed.) ESOP 2011. LNCS, vol. 6602, pp. 116–135. Springer, Heidelberg (2011); Full version as Microsoft Technical Report MSR-TR-2010-94
7. Decandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., Vogels, W.: Dynamo: amazon's highly available key-value store. In: Symposium on Operating Systems Principles, pp. 205–220 (2007)
8. Fekete, A., Liarokapis, D., O'Neil, E., O'Neil, P., Shasha, D.: Making snapshot isolation serializable. ACM Trans. Database Syst. 30(2), 492–528 (2005)
9. Gilbert, S., Lynch, N.: Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. SIGACT News 33, 51–59 (2002)
10. Gray, J., Helland, P., O'Neil, P., Shasha, D.: The dangers of replication and a solution. Sigmod Record 25, 173–182 (1996)
11. Herlihy, M., Koskinen, E.: Transactional boosting: a methodology for highly-concurrent transactional objects. In: Principles and Practice of Parallel Programming, PPoPP (2008)
12. Imine, A., Rusinowitch, M., Oster, G., Molli, P.: Formal design and verification of operational transformation algorithms for copies convergence. Theoretical Computer Science 351, 167–183 (2006)
13. Kaplan, H.: Persistent data structures. In: Handbook on Data Structures and Applications, pp. 241–246. CRC Press (1995)
14. Koskinen, E., Parkinson, M., Herlihy, M.: Coarse-grained transactions. In: Principles of Programming Languages, POPL (2010)
15. Petersen, K., Spreitzer, M., Terry, D., Theimer, M., Demers, A.: Flexible update propagation for weakly consistent replication. Operating Systems Review 31, 288–301 (1997)
16. Saito, Y., Shapiro, M.: Optimistic replication. ACM Computing Surveys 37, 42–81 (2005)
17. Shapiro, M., Kemme, B.: Eventual consistency. In: Encyclopedia of Database Systems, pp. 1071–1072 (2009)
18. Shapiro, M., Preguia, N., Baquero, C., Zawirski, M.: Conflict-free replicated data types (2011)
19. Sun, C., Ellis, C.: Operational transformation in real-time group editors: issues, algorithms, and achievements. In: Conference on Computer Supported Cooperative Work, pp. 59–68 (1998)
20. Terry, D., Theimer, M., Petersen, K., Demers, A., Spreitzer, M., Hauser, C.: Managing update conflicts in bayou, a weakly connected replicated storage system. SIGOPS Oper. Syst. Rev. 29, 172–182 (1995)
21. Valdes, J., Tarjan, R., Lawler, E.: The recognition of series parallel digraphs. In: ACM Symposium on Theory of Computing, pp. 1–12 (1979)