Declarative Serializable Snapshot Isolation

Christian Tilgner[1], Boris Glavic[2], Michael Böhlen[1], and Carl-Christian Kanne[3]

[1] University of Zurich,
{tilgner,boehlen}@ifi.uzh.ch
[2] University of Toronto,
glavic@cs.toronto.edu
[3] University of Mannheim
kanne@informatik.uni-mannheim.de

Abstract. Snapshot isolation (SI) is a popular concurrency control protocol, but it permits non-serializable schedules that violate database integrity. The Serializable Snapshot Isolation (SSI) protocol ensures (view) serializability by preventing pivot structures in SI schedules. In this paper, we leverage the SSI approach and develop the Declarative Serializable Snapshot Isolation (DSSI) protocol, an SI protocol that guarantees serializable schedules. Our approach requires no analysis of application programs or changes to the underlying DBMS. We present an implementation and prove that it ensures serializability.

1 Introduction

Snapshot Isolation (SI) [3] is a popular multiversion concurrency control (MVCC) protocol, but it permits non-serializable schedules. Fekete et al. [9] showed that every non-serializable SI schedule necessarily contains an access pattern with two consecutive vulnerable edges (see Sec. 2.2), and Cahill et al. [5] presented the *Serializable Snapshot Isolation (SSI)* protocol that ensures serializable schedules by preventing such structures.

We leverage the ideas of SSI, define *pivot structures* and propose the *Declarative Serializable Snapshot Isolation (DSSI)* protocol, a declarative technique that guarantees serializable schedules by preventing pivot structures while maintaining the advantages of SI. We implement DSSI using our declarative scheduling model called *Oshiya*. Oshiya models the scheduler state (including the generated schedule) in so-called *scheduling relations* and formalizes a protocol as a *protocol specification*. A protocol specification is a set of constraints specified as boolean domain relational calculus expressions that have to hold for all scheduling relation states. In Oshiya, a protocol specification is implemented as declarative *scheduling queries*. Request scheduling is performed by applying a generic scheduling algorithm that repeatedly executes the scheduling queries over the scheduling relations. The queries determine the pending requests that can be added to the relation modelling the schedule without violating the protocol specification. We show how to detect and prevent pivot structures using Oshiya and implement the DSSI protocol specification as scheduling queries. Our

J. Eder, M. Bielikova, and A.M. Tjoa (Eds.): ADBIS 2011, LNCS 6909, pp. 170–184, 2011.

implementation is concise and close to the formal protocol specification which enables us to prove its correctness. The main contributions of the paper are:

- We introduce DSSI, a protocol that ensures serializable SI executions, and formalize it as an Oshiya protocol specification.
- Using Oshiya we develop an SQL implementation of DSSI.
- We prove that the implementation ensures serializable schedules.

The paper structure is as follows: Sec. 2 describes SI and reviews the approach applied by the SSI protocol to detect non-serializable schedules. Sec. 3 introduces Oshiya. Sec. 4 shows how we model data snapshots and presents schemata for the scheduling relations. Sec. 5 formalizes the DSSI protocol. Sec. 6 presents the DSSI scheduler implementation. Sec. 7 proves that our implementation ensures serializable executions. We review related work in Sec. 8 and conclude in Sec. 9.

2 Background: Snapshot Isolation and Serializability

We model a transaction t_i as a sequence of read and write requests (denoted as $r_i(x)$ resp. $w_i(x)$ where x stands for the accessed data item). Each transaction finishes with an abort (a_i) or commit (c_i) request. The write-set WS_i of t_i contains all data items written by t_i. A *history* (schedule) is a sequence of interleaved executions of requests from a set of concurrent transactions. The requests in a history are totally ordered. We write $p <^H q$ if request p is executed before request q. Let bot_i denote the begin of t_i (when t_i executed its first request) and eot_i its end (when t_i aborted resp. committed). The execution interval of a committed transaction t_i is $[bot_i, c_i]$, the one of a non-aborted, possibly committed transaction t_i is $[bot_i, l_i]$ (l_i is t_i's latest operation). Two committed transactions t_i and t_j *overlapped* if: $Overlapped_{ij} \Leftrightarrow [bot_i, c_i] \cap [bot_j, c_j] \neq \emptyset$. Two non-aborted (maybe active) transactions t_i and t_j *overlap* if: $Overlap_{ij} \Leftrightarrow [bot_i, l_i] \cap [bot_j, l_j] \neq \emptyset$.

2.1 Snapshot Isolation

SI is a multiversion concurrency protocol that maintains multiple versions of data items (tuples). Each write $w_i(x)$ creates a new version of item x that is visible to other transactions after c_i. Each read $r_i(x)$ accesses the latest version of x written by transactions that committed before bot_i. Moreover, a transaction always sees the versions it created itself. Under SI, reads are never delayed because of write requests of concurrent transactions and vice versa. SI avoids inconsistent read anomalies because transactions never access partial results of other concurrent transactions. SI requires disjoint write-sets of concurrent committed transactions which is, e.g., ensured by the First-Committer-Wins (FCW) rule. FCW specifies that a transaction is aborted if a concurrent transaction with an overlapping write-set already committed. FCW also prevents lost updates. A typical anomaly that leads to non-serializable SI histories is the *Write Skew* [3], detailed in Ex. 1.

Example 1. Consider history H_{ws} in Fig. 1. Initially, data items $x = 50$ and $y = 50$ are consistent and satisfy constraint $C = x + y \geq 0$. Transaction t_1 reads

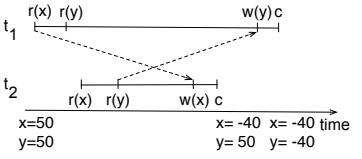

Fig. 1. History H_{ws}

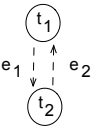

Fig. 2. MVSG for History H_{ws}

x and y. A concurrent transaction t_2 reads x and y, writes x (after subtracting 90) and commits (after checking C). Finally, t_1 writes y (after subtracting 90) and commits (after checking C). In the final state, C is violated although t_1 and t_2 checked C explicitly before committing. This happens because C is checked on the version of x and y that is visible to t_1 and t_2 and not on the final state resulting from their interleaved execution.

2.2 Detecting Non-serializable Histories

Serializability of SI histories can be checked using a multiversion serialization graph $MVSG = (N, E)$ [5]. The MVSG of a history H is a graph that contains a node for each committed transaction t_i of H: $t_i \in N \Leftrightarrow c_i \in H$. It contains an edge from transaction t_i to transaction t_j with $i \neq j$ if (a) $w_i(x) <^H w_j(x)$, (b) $w_i(x) <^H r_j(x)$, or (c) $r_i(x) <^H w_j(x)$. An edge of type (c) that occurs between two overlapped transactions t_i and t_j is called a *vulnerable edge* [9]. A *pivot structure* is defined as follows: $Overlapped_{ij} \wedge (r_i(x) <^H w_j(x)) \wedge Overlapped_{jk} \wedge (r_j(x) <^H w_k(x))$. Fekete et al. [9] showed that every MVSG of a non-serializable SI history must contain a pivot structure. The existence of a pivot structure is a necessary but not sufficient condition for the non-serializability of an SI history. Thus, an SI history is serializable if its MVSG does not contain pivot structures.

Example 2. Fig. 1 shows history H_{ws}. Vulnerable edges are shown as dotted lines. The MVSG for H_{ws} in Fig. 2 has a node for each committed transaction of H_{ws} (t_1 and t_2) and two edges e: (e_1) from t_1 to t_2 due to $r_1(x) <^H w_2(x)$; (e_2) from t_2 to t_1 due to $r_2(y) <^H w_1(y)$. H_{ws} is not serializable and, thus, the MVSG contains a pivot structure (two consecutive vulnerable edges e_1 and e_2).

2.3 Serializable Snapshot Isolation Protocol

The SSI protocol proposed by Cahill et al. [5] ensures serializability by preventing pivot structures. The main idea is to check SI histories at runtime for structures that can evolve into pivot structures. We call such structures *potential pivot structures*. A potential pivot structure is defined as: $Overlap_{ij} \wedge (r_i(x) <^H w_j(x)) \wedge Overlap_{jk} \wedge (r_j(x) <^H w_k(x)) \wedge \neg(c_i \wedge c_j \wedge c_k)$. I.e., a potential pivot structure is a pivot structure without the requirement that the three (not necessarily distinct) participating transactions have committed. It evolves into a pivot structure once all participating transactions have committed. The set of transactions in potential pivot structures is naturally a superset of the transactions in pivot structures. For each detected potential pivot structure, one of

the participating transactions is aborted to prevent it from evolving into a pivot structure. This approach guarantees that the resulting histories are serializable, but it may produce false positives, i.e., not every potential pivot structure finally results in a non-serializable history. Our implementation leverages this idea and aborts transactions that participate in potential pivot structures (see Sec. 6).

3 Declarative Scheduling Model

We propose a *declarative scheduling model* [13] called *Oshiya*[1] to model and implement DSSI. The main ideas of Oshiya are: (1) The state of a scheduler (including the history it produces) is modeled as instances of three *scheduling relations*: *PendingRequests* ($\mathcal{R}$) buffers arriving client requests for scheduling. *RelevantHistory* ($\mathcal{H}$) stores already executed requests in their execution order, and models the schedule generated so far. *Executable* ($\mathcal{E}$) buffers requests that have been scheduled for execution. (2) Oshiya formalizes a protocol as a set of constraints, called *protocol specification*, that have to hold for each generated state of $\mathcal{H}$. (3) The protocol specification constraints are implemented as declarative *scheduling queries*: $Q_{Schedule}$, $Q_{Revoked}$, $Q_{Irrelevant}$. Request scheduling is performed by repeatedly executing the *scheduling queries* over the *scheduling relations* to determine which of the pending requests in $\mathcal{R}$ can be added to $\mathcal{H}$ without violating the protocol specification constraints.

Example 3. For presentation purposes, we use simplified schemata for the scheduling relations in this example. Assume the following schema for relations $\mathcal{R}$ and $\mathcal{E}$: (TA, Op, Ob). For each request, TA is the transaction executing the request, Op is the type of operation (e.g., r for a read), and Ob is the data object the operation accesses. Relation $\mathcal{H}$ has an additional attribute ID for recording the request execution order. Using this schema, the scheduler state after scheduling the first request from history H_{ws} (Fig. 1) is as follows:

$\mathcal{R}_1$

TA	Op	Ob	...
1	r	x	

$\mathcal{H}_1$

ID	TA	Op	Ob	...
1	1	r	x	

$\mathcal{E}_1$

TA	Op	Ob	...
1	r	x	

The state of the scheduler is advanced in iterative steps by applying a generic scheduling algorithm (shown on the right) that evaluates the scheduling queries over the current instances of the scheduling relations. Each iterative step (one while loop), called *scheduler iteration*, schedules multiple requests at once, resulting in updated instances of the scheduling relations. This is in contrast to DBMSs that schedule requests individually. The algorithm is the same for every protocol, but it is parameterized by the protocol specific schema of the scheduling

```
1  H = E = R = ∅
2  while true do begin
3        R = R − E;
4        R = R ∪ N;
5        R = R − Q_Revoked(H, R);
6        E = Q_Schedule(H, R);
7        Execute(E);
8        H = H ∪ E;
9        H = H − Q_Irrelevant(H);
10 end
```

[1] Oshiya refers to the passenger arrangement staff at Japanese train stations who help to fill a train by pushing people onto the train or guiding them to free railway cars.

relations and the scheduling queries. $\mathcal{N}$ is the set of newly arrived client requests. $Q_{Revoked}$ identifies nonexecutable requests (e.g., deadlocked) (line 5). $Q_{Schedule}$, the main scheduling query, identifies the pending requests from $\mathcal{R}$ that should be selected for execution in this iteration (line 6). $Q_{Irrelevant}$ returns requests that are irrelevant for future scheduling decisions. They are removed from $\mathcal{H}$ (line 9). In the remainder of this paper we limit the discussion to $Q_{Schedule}$.

Example 4. Reconsider the scheduler state from Ex. 3. Two new requests got inserted into $\mathcal{R}$ at the beginning of scheduler iteration 2: $r_1(y)$, $r_2(x)$. Assume that running the scheduling queries selected both request from $\mathcal{R}$ for execution. This leads to the following updated scheduler state:

$\mathcal{R}_2$

TA	Op	Ob	...
1	r	y	
2	r	x	

$\mathcal{H}_2$

ID	TA	Op	Ob	...
1	1	r	x	
2	1	r	y	
3	2	r	x	

$\mathcal{E}_2$

TA	Op	Ob	...
1	r	y	
2	r	x	

Applying the scheduling queries to a set of newly arrived requests $\mathcal{N}$, each scheduler iteration produces new instances of the scheduling relations $\mathcal{R}$, $\mathcal{H}$ and $\mathcal{E}$. This yields a sequence of states of $\mathcal{H}$ called history, defined below. We use this definition of history to reason over the properties of a protocol and to prove the correctness of a scheduler implementation.

Definition 1 (History). *Let $\mathbb{I} =< \mathcal{N}_0, \ldots >$ be a sequence of sets of input requests. Let q be protocol-specific versions of the scheduling queries. We define the history $\mathbb{H}_q(\mathbb{I})$ generated according to q over input $\mathbb{I}$ as $< \mathcal{H}_0, \ldots >$, where $\mathcal{H}_i$, called a history state, is the state of relation $\mathcal{H}$ after the i^{th} scheduler iteration produced using q to parameterize the generic algorithm and $\mathcal{N}_i$ as input $\mathcal{N}$. In the paper, we drop q and $\mathbb{I}$ if it is clear from the context and solely use $\mathbb{H}$.*

In the remainder of this paper, we use $\mathcal{H}$ to denote both the history relation and one history state and drop indices on $\mathcal{H}$ if the scheduler iteration is irrelevant for the discussion (same holds for $\mathcal{R}$, $\mathcal{E}$ and $\mathcal{N}$). According to the algorithm presented above, the history state $\mathcal{H}_i$ is a *cumulative snapshot*, i.e., it includes all previous history states $\mathcal{H}_j$ with $j < i$.

Example 5. For instance, the history states shown below could be the result of scheduling the requests $\mathbb{I} =< \{(1, r, x), (2, r, x)\}, \{(1, r, y)\}, \{(2, r, y)\} >$:

$\mathcal{H}_0$

ID	TA	Op	Ob	...

$\mathcal{H}_1$

ID	TA	Op	Ob	...
1	1	r	x	

$\mathcal{H}_2$

ID	TA	Op	Ob	...
1	1	r	x	
2	1	r	y	
3	2	r	x	

$\mathcal{H}_3$

ID	TA	Op	Ob	...
1	1	r	x	
2	1	r	y	
3	2	r	x	
4	2	r	y	

We model a protocol as a set of constraints called *protocol specification*. A protocol specification constraint is a boolean *domain relational calculus* expression over histories. We allow quantification over scheduler iterations to enable, e.g., constraints that check the order of requests in the history.

Definition 2 (Protocol Specification). *A protocol specification Φ is a set of boolean domain relational calculus expressions over $\mathbb{H}$.*

The formalization of a protocol as logical constraints and its implementation as queries allows us to formally reason about the correctness of an implementation. Given a protocol specification Φ and an implementation of this protocol as a set q of scheduling queries, the definition presented below defines what it means for q to correctly implement Φ. Intuitively, this is the case if for every input $\mathbb{I}$, the history created by our scheduling algorithm using q satisfies Φ. We use this definition in Sec. 7 to prove the correctness of our DSSI implementation.

Definition 3 (Correctness of Scheduling Queries). *Scheduling queries q satisfy a protocol specification Φ, denoted as $q \models \Phi$, if for every input sequence $\mathbb{I}$ the generated history $\mathbb{H}$ produced using q satisfies Φ: $\mathbb{H}_q(\mathbb{I}) \models \Phi$.*

3.1 Assumptions and Notational Remarks

We make the following assumptions: (1) Client requests read resp. manipulate only one tuple. (2) A transaction waits until its current request is executed before issuing new requests. (3) Object identifiers are unique over all relations. (4) Rollbacks of transactions are considered as regular requests issued by clients. Extending Oshiya to schedule complex queries like joins or range queries is an interesting avenue for future work. Assumptions 2-4 simplify the presentation, but can be changed with minor modifications to Oshiya.

Scheduling queries and protocol specifications are given as domain relational calculus expressions. Capital letters denote variables, small letters indicate constants and ϵ denotes *null*. All non-target variables not used in a universal quantification are implicitly existentially quantified. E.g., instead of $\{A \mid \exists B : (I(A,B) \wedge \neg\exists C : (J(C,A)))\}$ we write $\{A \mid I(A,B) \wedge \neg J(C,A)\}$. Unrestricted existentially quantified variables are displayed as an underline ("_"), disjunctive use of constants by "|". E.g., for the expression $I(A,B) \wedge (A = a \vee A = c)$ we use the shortcut $I(a|c,\_)$. We define aggregation as: $\{G, F_1(A_1), \ldots, F_n(A_n) \mid E\}$. E is a domain relational calculus expression, G is a set of attributes on which to group on (can be empty), and each F_i is an aggregate over attribute A_i.

4 Modeling Data Relation Snapshots and Defining the Oshiya Scheduling Relation Schemata for DSSI

In order to implement DSSI with Oshiya, we have to (1) specify the schema of the scheduling relations that model the scheduler state, (2) formalize the protocol specification based on these relations (Sec. 5), and (3) implement the protocol specification as scheduling queries (Sec. 6). In this section, we show how to adapt data relation schemata to support data item versions (Sec. 4.1) and develop protocol-specific schemata for the Oshiya scheduling relations (Sec. 4.2).

4.1 Modeling Snapshots with Data Relations

We model snapshots explicitly by extending the schemata of data relations. This allows us to achieve DB independence and to run DSSI on DBMSs that do not

support snapshots. We identify a version of data item x using a tuple (TA, Seq) where TA is the transaction that created the version and Seq is the position of the request within this transaction. Of course, versions can be modeled differently but this is orthogonal to our approach and beyond the scope of this paper. Given a database schema with relations $R_1, \ldots, R_n$, we map each relation R_i to a relation R_i' which has four additional attributes. These attributes store the version identifier for the creator transaction (CTA and $CSeq$) and, if applicable, for the transaction that deleted the data item (DTA and $DSeq$). The primary key of R_i' is the primary key of R_i union the attributes CTA and $CSeq$.

Example 6. Assume a bank stores account data with account numbers and balances in relation *Accounts*(AccNr,Bal). We map this relation to *Accounts*′ by extending its schema with the four additional attributes mentioned above. An example instance shown on the right contains an initial version of object x created by transaction t_1 ($CTA = 1, CSeq = 1$) and two new versions created by t_2 and t_3.

Accounts′

AccNr	Bal	CTA	CSeq	DTA	DSeq
x	5	1	1	-	-
x	10	2	2	-	-
x	15	3	1	-	-

4.2 Oshiya Scheduling Relation Schemata

For DSSI, we use the schemata for *scheduling relations* $\mathcal{R}$, $\mathcal{H}$ and $\mathcal{E}$ shown below. For simplicity, we present only attributes needed for scheduling and omit those necessary for request execution (e.g., the value to be written for write requests).

$\mathcal{R}$ (TA,Seq,Op,OID) $\mathcal{H}$ (ID,TA,Seq,Op,OID,OTA,OSeq) $\mathcal{E}$ (ID,TA,Seq,Op,OID,OTA,OSeq)

For each incoming request, we insert a tuple into $\mathcal{R}$ storing an identifier T_i for the transaction t_i that issued the request (TA), the request position within this transaction (Seq), the type of operation (read, write, abort or commit, stored in attribute Op) and the data object the requests is applied to (OID). Transactions identifiers (TA) are ordered, i.e., if $bot_i < bot_j$ then $T_i < T_j$. $\mathcal{H}$ and $\mathcal{E}$ contain additional attributes: ID records the execution order of requests. For read requests, OTA and $OSeq$ store which object version was read by the request. These attributes correspond to the data relations attributes CTA and $CSeq$.

Example 7. Assume the instances of relations $\mathcal{R}$ and $\mathcal{H}$ displayed below. $\mathcal{H}$ contains the requests that produced the state of relation *Accounts*′ from Ex. 6: (1) and (2) Transaction t_1 created the initial version of object x and committed. (3) Transaction t_2 read this version of object x. (4) and (5) t_2 and t_3 wrote new versions of object x. (6) t_2 committed. (7) t_4 read the new version created by t_2. At this iteration, $\mathcal{R}$ contains no pending requests that have to be scheduled.

$\mathcal{R}$

TA	Seq	Op	Ob

$\mathcal{H}$

ID	TA	Seq	Op	Ob	OTA	OSeq
1	1	1	w	x	-	-
2	1	2	c	-	-	-
3	2	1	r	x	1	1
4	2	2	w	x	-	-
5	3	1	w	x	-	-
6	2	3	c	-	-	-
7	4	1	r	x	2	2

5 DSSI Protocol Specification

We now develop the protocol specification for DSSI based on the scheduling relations presented in Sec. 4. Recall from Sec. 3 that a protocol specification models a protocol as a set of domain relational calculus expressions over histories.

To formalize SI with Oshiya, we use views over relation $\mathcal{H}$ to get the relevant information described in Sec. 2. For *bot*, we use view $BOT(TA, ID)$ querying for each transaction (TA) the ID of its first request in $\mathcal{H}$. $EOT(TA, Op, ID)$ selects for each finished transaction t_i (TA) the ID of its final request in $\mathcal{H}$ (corresponds to eot_i) and whether t_i aborted or committed (Op). $Overlap(TA1, TA2)$ contains all pairs of concurrently executed, non-aborted transactions, i.e., they do not have to be committed. $PotPivotStr(TA1, TA2, TA3)$ selects all triples of transactions forming potential pivot structures as described in Sec. 2.3.

C1 (Read Versions). The SI protocol specifies [3,5,14] that a read request $r_i(x)$ of a transaction t_i reads t_i's most recent changes to x. If no such changes exist, then $r_i(x)$ reads the latest version of x created by transactions that committed before t_i started. These conditions are formalized as protocol specification constraint C1 (a) and (b) shown in Fig. 3: **(a)** The first case applies if a transaction T has written object O before reading a version (X, Y) of O:

$$\mathcal{H}(I, T, N, r, O, X, Y) \wedge \mathcal{H}(I_2, T, N_2, w, O, \_, \_) \wedge I_2 < I$$

It follows that T read a version it created itself $(X = T)$ and (X, Y) is the latest version produced by T before the read (no newer versions exist):

$$X = T \wedge N_2 = Y \wedge \neg(\mathcal{H}(\_, T, N_2, w, O, \_, \_) \wedge Y < N_2 < N)$$

(b) The second case applies if T has not written O before the read was executed: $\neg(\mathcal{H}(I_2, T, \_, w, O, \_, \_) \wedge I_2 < I)$. It follows that (1) O was written by another transaction X and X committed before T started. (2) (X, Y) has to be the latest version written by X and (3) there may not be another version written by a transaction T_2 that committed after X but before T started:

(**1**) $X \neq T \wedge EOT(X, c, I_3) \wedge BOT(T, I_4) \wedge I_3 < I_4$ (**2**) $\neg(\mathcal{H}(\_, X, N_3, w, O, \_, \_) \wedge N_3 > Y)$
(**3**) $\neg(\mathcal{H}(\_, T_2, \_, w, O, \_, \_) \wedge EOT(T_2, c, I_5) \wedge I_4 < I_5 < I_3)$

C2 (FCW). SI requires disjoint write-sets for all committed concurrent transactions. Protocol specification constraint C2 (see Fig. 3) models this condition as follows. If (1) two overlapping transactions T and T_2 (2) both wrote the same object O and (3) T did already commit, then (4) T_2 did not commit:

(**1**) $Overlap(T, T_2)$ (**2**) $\mathcal{H}(\_, T, \_, w, O, \_, \_) \wedge \mathcal{H}(\_, T_2, \_, w, O, \_, \_)$
(**3**) $EOT(T, c, \_)$ (**4**) $\neg EOT(T_2, c, \_)$

C3 (Serializability). Recall that an SI history is serializable, if it does not contain pivot structures. In constraint C3 (see Fig. 3), we follow the approach outlined in Sec. 2.3: If (1) relation $\mathcal{H}$ contains a potential pivot structure, then we require that (2) at least one of the participating transactions did not commit:

(**1**) $PotPivotStr(T, T_2, T_3)$ (**2**) $\neg(EOT(T, c, \_) \wedge EOT(T_2, c, \_) \wedge EOT(T_3, c, \_))$

(C1) **(a)** $\forall I, N, O, T, X, Y : \mathcal{H}(I, T, N, r, O, X, Y) \wedge \mathcal{H}(I_2, T, N_2, w, O, \_, \_) \wedge I_2 < I \Rightarrow$
$X = T \wedge N_2 = Y \wedge \neg(\mathcal{H}(\_, T, N_3, w, O, \_, \_) \wedge Y < N_3 < N)$
(b) $\forall I, N, O, T, X, Y : \mathcal{H}(I, T, N, r, O, X, Y) \wedge \neg(\mathcal{H}(I_2, T, \_, w, O, \_, \_) \wedge I_2 < I) \Rightarrow$
$X \neq T \wedge EOT(X, c, I_3) \wedge BOT(T, I_4) \wedge I_3 < I_4 \wedge \neg(\mathcal{H}(\_, X, N_3, w, O, \_, \_) \wedge N_3 > Y) \wedge$
$\neg(\mathcal{H}(\_, T_2, \_, w, O, \_, \_) \wedge EOT(T_2, c, I_5) \wedge I_3 < I_5 < I_4)$

(C2) $\forall O, T, T_2 : Overlap(T, T_2) \wedge \mathcal{H}(\_, T, \_, w, O, \_, \_) \wedge \mathcal{H}(\_, T_2, \_, w, O, \_, \_) \wedge EOT(T, c, \_)$
$\Rightarrow \neg EOT(T_2, c, \_)$

(C3) $\forall T, T_2, T_3 : PotPivotStr(T, T_2, T_3) \Rightarrow \neg(EOT(T, c, \_) \wedge EOT(T_2, c, \_) \wedge EOT(T_3, c, \_))$

Fig. 3. DSSI Protocol Specification

6 DSSI Implementation

Recall that with Oshiya, protocols are implemented as scheduling queries. We implemented all scheduling queries for DSSI, but in this paper we only describe $Q_{Schedule}$. Our prototype implementation of Oshiya requires the scheduling queries to be expressed in SQL. However, for conciseness, domain relational calculus expressions are used throughout this section. $Q_{Schedule}$ is developed in two steps. First we present queries necessary to detect potential pivot structures (Sec. 6.1). Afterwards, we use these queries to implement $Q_{Schedule}$ (Sec. 6.2). Recall that detecting potential pivot structures and aborting one of the participating transactions ensure serializability. However, this approach may detect false positives (see Sec. 2). Studying the trade-off between the number of false positives and the cost of scheduling is an interesting avenue for future work.

6.1 Detecting Potential Pivot Structures

We now discuss how to express *BOT*, *EOT*, *Overlap* and *PotPivotStr* introduced in Sec. 5 as queries over $\mathcal{H}$. *BOT* and *EOT* are defined below. E.g., *EOT* queries for each finished transaction T its abort resp. commit state (A) and its *eot* (I) which is equal to the ID of its abort resp. commit request in $\mathcal{H}$.

$$BOT = \{T, I \mid \mathcal{H}(I, T, \_, \_, \_, \_, \_) \wedge \neg(\mathcal{H}(I_2, T, \_, \_, \_, \_, \_) \wedge I_2 < I)\}$$
$$EOT = \{T, A, I \mid \mathcal{H}(I, T, \_, A, \_, \_, \_) \wedge A = a|c\}$$

Overlapping transactions are inferred as specified below. Two (1) non-aborted transactions T_1 and T_2 overlap if (2) $bot_1 <^H bot_2$ and (3) $bot_2 <^H c_1$ (if T_1 has already committed) or (4) the symmetric case holds:

$$Overlap = \{T_1, T_2 \mid T_1 \neq T_2 \wedge \neg EOT(T_1|T_2, a, \_) \wedge \quad (1)$$
$$((BOT(T_1, I) \wedge BOT(T_2, I_2) \wedge I < I_2 \wedge \quad (2)$$
$$(EOT(T_1, c, I_3) \Rightarrow I_2 < I_3)) \vee \quad (3)$$
$$(BOT(T_2, I_2) \wedge BOT(T_1, I) \wedge I_2 < I \wedge (EOT(T_2, c, I_3) => I < I_3)))\} \quad (4)$$

We use *PotVulnEdge* to query all potential vulnerable edges between concurrent, non-aborted transactions T and T_2 (potential, because T and T_2 might not yet have committed). *PotPivotStr* detects potential pivot structures by checking for transactions (T_2) that have both an incoming and outgoing *PotVulnEdge*:

$$PotVulnEdge = \{T, T_2 \mid \mathcal{H}(I, T, \_, r, O, \_, \_) \wedge \mathcal{H}(I_2, T_2, \_, w, O, \_, \_) \wedge Overlap(T, T_2) \wedge I < I_2\}$$
$$PotPivotStr = \{T, T_2, T_3 \mid PotVulnEdge(T, T_2) \wedge PotVulnEdge(T_2, T_3)\}$$

Example 8. We show the results of the queries defined above (highlighted) for the history state $\mathcal{H}$ from Ex. 7. For instance, *PotVulnEdge* contains one potential vulnerable edge from transaction t_2 to t_3, because t_2 and t_3 overlap and t_2 read object x and afterwards t_3 wrote a new version of object x ($r_2(x) <^H w_3(x)$).

$\mathcal{H}$

ID	TA	Seq	Op	Ob	OTA	OSeq
1	1	1	w	x	-	-
2	1	2	c	-	-	-
3	2	1	r	x	1	1
4	2	2	w	x	-	-
5	3	1	w	x	-	-
6	2	3	c	-	-	-
7	4	1	r	x	2	2

BOT

TA	ID
1	1
2	3
3	5
4	7

EOT

TA	Op	ID
1	c	2
2	c	6

Overlap

TA1	TA2
2	3
3	2
3	4
4	3

PotVulnEdge

TAout	TAin
2	3

PotPivotStr

TA1	TA2	TA3

6.2 $Q_{Schedule}$

The DSSI version of $Q_{Schedule}$ implementing the protocol specification constraints C1-C3 is shown in Fig. 4. According to the SI conditions, all write, abort, and read requests from $\mathcal{R}$ may always be selected for execution. $Q_{Schedule}$ selects all of these requests using queries *AbortWrites* and *Reads*. Which commit requests can be selected without violating constraints $C2$ and $C3$ is determined through query *ValidCommits*. In $Q_{Schedule}$, function *GenID()* generates unique values for the *ID* attribute of $\mathcal{H}$ (modelling the execution order of requests).

Read Requests (C1). The *Reads* query uses *LVV* (last valid version) to select for each read request of transaction T on object O the version (T_2, N_2) that has to be read. Recall that attributes *OTA* and *OSeq* of relations $\mathcal{E}$ and $\mathcal{H}$ identify a version of an object O. Version (T_2, N_2) is computed in two steps. *LastOTA* queries the transaction identifier (T_2) of the transaction that wrote the version of O that has to be read by T. Based on this information *LVV* determines N_2, the *Seq* value of the latest write request of T_2 on object O. T_2 is the maximal value from the following union: (a) T_2=T if T itself created versions of O and (b) transactions that wrote a version of O and committed before T started.

$$\textbf{(a)}\ \mathcal{H}(\_, T_2, \_, w, O, \_, \_) \wedge T = T_2 \quad \textbf{(b)}\ \mathcal{H}(\_, T_2, \_, w, O, \_, \_) \wedge EOT(T_2, c, I_2) \wedge (BOT(T, I) \Rightarrow I_2 < I)$$

Example 9. Consider $\mathcal{H}$ from Ex. 8. $r_2(x)$ read the initial version of object x (since $c_1 <^H bot_2$) and $r_4(x)$ read the version written by t_2 (since $c_2 <^H bot_4$).

Commit Requests (C2 and C3). To guarantee that constraints C2 and C3 hold for each history produced by $Q_{Schedule}$, we have to prevent commit requests to be executed if (1) the commit would violate the FCW rule (C2) or (2) the commit would violate serializability (C3). There are two possible ways how the execution of commit requests can violate the FCW rule: (1a) A commit is from a transaction whose write-set overlaps with the one of a concurrent but already committed transaction and (1b) if $\mathcal{R}$ contains commit requests from multiple

$$
\begin{aligned}
Q_{Schedule} &= \{GenID(), T, N, A, O, T_2, N_2 \mid \mathcal{R}(T, N, A, O) \wedge (ValidCommits(T, N, T_2, N_2) \\
&\quad \vee AbortsWrites(T, N, T_2, N_2) \vee Reads(T, N, T_2, N_2))\} \\
\hline
AbortsWrites &= \{T, N, \epsilon, \epsilon \mid \mathcal{R}(T, N, a|w, \_)\} \\
\hline
Reads &= \{T, N, T_2, N_2 \mid \mathcal{R}(T, N, r, O) \wedge LVV(T, O, T_2, N_2)\} \\
LVV &= \{T, O, T_2, MAX(N_2) \mid LastOTA(T, O, T_2) \wedge \mathcal{H}(\_, T_2, N_2, w, O, \_, \_)\} \\
LastOTA &= \{T, O, MAX(T_2) \mid \mathcal{R}(T, \_, r, O) \wedge ((\mathcal{H}(\_, T_2, \_, w, O, \_, \_) \wedge T = T2) \vee \\
&\quad (\mathcal{H}(\_, T_2, \_, w, O, \_, \_) \wedge EOT(T_2, c, I_2) \wedge (BOT(T, I) \Rightarrow I_2 < I)))\} \\
\hline
ValidCommits &= \{T, N, \epsilon, \epsilon \mid NonForbCs(T, N) \wedge \neg DelayedCs(T, N)\} \\
DelayedCs &= \{T, N \mid NonForbCs(T, N) \wedge NonForbCs(T_2, \_) \wedge \\
&\quad \mathcal{H}(\_, T, \_, w, O, \_, \_) \wedge \mathcal{H}(\_, T_2, \_, w, O, \_, \_) \wedge T > T_2\} \\
NonForbCs &= \{T, N \mid \mathcal{R}(T, N, c, \_) \wedge \neg(ForbCs(T, N) \vee ForbCinPPS(T, N))\} \\
ForbCinPPS &= \{T, N \mid \mathcal{R}(T, N, c, \_) \wedge PotPivotStr(T_2, T_3, T_4) \wedge (T = T_2|T_3|T_4) \wedge \\
&\quad \neg(\mathcal{R}(T_5, \_, c, \_, \_) \wedge (T_5 = T_2|T_3|T_4) \wedge T < T_5)\} \\
ForbCs &= \{T, N \mid \mathcal{R}(T, N, c, \_) \wedge \mathcal{H}(\_, T, \_, w, O, \_, \_) \wedge \mathcal{H}(\_, T_2, \_, w, O, \_, \_) \wedge \\
&\quad Overlap(T, T_2) \wedge EOT(T_2, c, \_)\}
\end{aligned}
$$

Fig. 4. $Q_{Schedule}$

transactions with overlapping write-sets, then only one of these transaction may commit. Note that in the concrete implementation, commits identified to violate C2 or C3 are selected by $Q_{Revoked}$ and aborted.

We use a two stage approach to select valid commits: In step 1, query *NonForbCs* selects commits from $\mathcal{R}$ and filters out commits of case 1a using query *ForbCs* and those of case 2 using query *ForbCinPPS*. *NonForbCs* may still contain sets of commit requests from transactions with overlapping write-sets (case 1b). We only allow the oldest transaction from each set to commit. Therefore, in step 2, query *ValidCommits* selects all requests from *NonForbCs* and uses query *DelayedCs* to keep only the commit request of the oldest transaction for each set of transactions with overlapping write-sets.

Step 1. Query *ForbCs* (case 1a) identifies commits of transactions T that (a) wrote an object also written by an (b) overlapping committed transaction T_2.

(**a**) $\mathcal{H}(\_, T, \_, w, O, \_, \_) \wedge \mathcal{H}(\_, T_2, \_, w, O, \_, \_)$ (**b**) $Overlap(T, T_2) \wedge EOT(T_2, c, \_)$

ForbCinPPS (case 2) selects a commit of transaction T from $\mathcal{R}$ if (a) T belongs to potential pivot structure p and (b) $\mathcal{R}$ does not contain a commit request of a younger transaction T_5 (recall that $bot_1 < bot_2 \Rightarrow T_1 < T_2$) also belonging to p. Thus, if $\mathcal{R}$ contains commits of more than one of the transactions belonging to p, we disallow only the youngest one to commit (and abort it using $Q_{Revoked}$).

(**a**) $PotPivotStr(T_2, T_3, T_4) \wedge (T = T_2|T_3|T_4)$ (**b**) $\neg(\mathcal{R}(T_5, \_, c, \_, \_) \wedge (T_5 = T_2|T_3|T_4) \wedge T < T_5)$

Example 10. Consider the instances of $\mathcal{R}$ and $\mathcal{H}$ shown below that model history H_{ws} from Fig. 1. To keep the example simple, we do not show the actions of transaction t_0 that created the initial versions of objects x and y. Requests c_1 and c_2 belong to the same potential pivot structure p. Their execution can lead to a write skew violating C3. $Q_{Schedule}$ selects c_1 (smallest TA value). c_2 (commit of youngest transaction) is selected by *ForbCinPPS* and aborted to break p.

$\mathcal{R}$

TA	Seq	Op	Ob
1	4	c	-
2	4	c	-

$Q_{Schedule}$

ForbCs	DelayedCs	ForbCinPPS
X		
		X

$\mathcal{H}$

ID	TA	Seq	Op	Ob	OTA	OSeq
1	1	1	r	x	0	1
2	1	2	r	y	0	2
3	2	1	r	x	0	1
4	2	2	r	y	0	2
5	1	3	w	x	-	-
6	2	3	w	y	-	-

Overlap

TA1	TA2
1	2
2	1

PotVulnEdge

TAout	TAin
2	1
1	2

PotPivotStr

TA1	TA2	TA3
1	2	1
2	1	2

Step 2. *DelayedCs* detects case 1b by selecting all transactions T from *NonForbCs* where (a) *NonForbCs* contains another transaction T_2 which (b) wrote an object O that has also been written by T and (c) which is older than T.

(**a**) $NonForbCs(T_2, \_)$ (**b**) $\mathcal{H}(\_, T, \_, w, O, \_, \_) \wedge \mathcal{H}(\_, T_2, \_, w, O, \_, \_)$ (**c**) $T > T_2$

Example 11. Consider the instances of $\mathcal{R}$ and $\mathcal{H}$ displayed below. $Q_{Schedule}$ selects all read ($r_6(x)$) and write ($w_7(y)$) requests. c_3 belongs to *ForbCs* because transaction t_3 wrote the same object as the concurrent but already committed transaction t_2 and is, thus, not allowed to commit. c_4 and c_5 belong to *NonForbCs*, but t_4 and t_5 both wrote the same object x. *ValidCommits* selects only c_4 (oldest transaction from the set $\{t_4, t_5\}$ of transactions with overlapping write-set). c_5 is filtered out by *DelayedCs*.

$\mathcal{R}$

TA	Seq	Op	Ob
3	2	c	-
4	3	c	-
5	2	c	-
6	1	r	x
7	1	w	y

$Q_{Schedule}$

ValidCommits	DelayedCs	NonforbCs	ForbCinPPS	ForbCs
				X
X		X		
	X	X		
X				
X				

$\mathcal{H}$

ID	TA	Seq	Op	Ob	OTA	OSeq
1	1	1	w	x	-	-
2	1	2	c	-	-	-
3	2	1	r	x	1	1
4	2	2	w	x	-	-
5	3	1	w	x	-	-
6	2	3	c	-	-	-
7	4	1	r	x	2	2
8	4	2	w	x	-	-
9	5	1	w	x	-	-

7 Correctness Analysis

We now proof that every history produced under DSSI is serializable. Recall that an SI history is serializable if it does not contain a pivot structure. Thus, we can show this fact by proving that $\mathcal{H}$ cannot contain a potential pivot structure between committed transactions (equivalent after Sec. 2.3). Note that the influence of the other scheduling queries (mentioned in Sec. 3) on the results of $Q_{Schedule}$ and the compliance of C1 and C2 are not in the scope of this paper.

Theorem 1 ($Q_{Schedule}$ Prevents Pivot Structures). $Q_{Schedule} \models C3$

Proof. We omit to prove that the query *PotPivotStr* returns all potential pivot structures contained in $\mathcal{H}$, because the proof is trivial. We proof Theorem 1 by contradiction. Assume the negation of C3 holds:

$$\neg(\forall T, T_2, T_3 : PotPivotStr(T, T_2, T_3) \Rightarrow \neg(EOT(T, c, \_) \wedge EOT(T_2, c, \_) \wedge EOT(T_3, c, \_)))$$
$$\Leftrightarrow \quad \exists T, T_2, T_3 : \quad PotPivotStr(T, T_2, T_3) \wedge EOT(T, c, \_) \wedge EOT(T_2, c, \_) \wedge EOT(T_3, c, \_)$$

Let k be the first scheduler iteration where this equation holds for a fixed T_1, T_2, T_3 and T_4.

$$\Leftrightarrow \exists T, T_2, T_3, k : PotPivotStr_k(T, T_2, T_3) \wedge EOT_k(T, c, \_) \wedge EOT_k(T_2, c, \_) \wedge EOT_k(T_3, c, \_)$$

Without loss of generality, let T_3, the transaction at the third position of the potential pivot structure ($PotPivotStr(T, T_2, T_3)$), be the youngest transaction of the participating transactions. This assumption does not result in a loss of generality, because the position of T_3 is irrelevant for the rest of the proof. There must exist a scheduler iteration $i < k$ where T_3 has not yet committed but already belongs to *PotPivotStr*.

$$\Rightarrow \exists i : PotPivotStr_i(T, T_2, T_3) \wedge T_3 > T \wedge T_3 > T_2 \wedge \neg EOT_i(T_3, c, \_)$$

It follows that the commit request c_3 of T_3 occurs in relation $\mathcal{R}$ at some scheduler iteration j ($i < j < k$). To be executed, c_3 has to belong to the set of non-forbidden commits (*NonForbCs*). We can assume $PotPivotStr_i(T, T_2, T_3) \Rightarrow PotPivotStr_j(T, T_2, T_3)$.

$$\Rightarrow \exists j : PotPivotStr_j(T, T_2, T_3) \wedge T_3 > T \wedge T_3 > T_2 \wedge \neg EOT_j(T_3, c, \_) \wedge NonForbCs_j(T_3, \_)$$

We now replace *NonForbCs* by its definition and, afterwards, remove terms that are not needed to derive the contradiction:

$$\begin{aligned}
&\Leftrightarrow \exists j : PotPivotStr_j(T, T_2, T_3) \wedge T_3 > T \wedge T_3 > T_2 \wedge \neg EOT_j(T_3, c, _) \wedge \\
&\qquad \mathcal{R}_j(T_3, _, c, _) \wedge \neg ForbCs_j(T_3, _) \wedge \neg ForbCinPPS_j(T_3, _) \\
&\Rightarrow \exists j : PotPivotStr_j(T, T_2, T_3) \wedge T_3 > T \wedge T_3 > T_2 \wedge \mathcal{R}_j(T_3, _, c, _) \wedge \neg ForbCinPPS_j(T_3, _)
\end{aligned}$$

Since c_3 in $\mathcal{R}$ is the commit request of the youngest transaction participating in p, $\mathcal{R}$ cannot contain a commit request of a transaction that is both younger than T_3 and also belongs to p:

$$\begin{aligned}
&\Leftrightarrow \exists j : \mathcal{R}_j(T_3, _, c, _) \wedge PotPivotStr_j(T, T_2, T_3) \wedge \neg(\mathcal{R}_j(T_4, _, c, _) \wedge T_4 = T|T_2 \wedge T_4 < T_3) \wedge \\
&\qquad \neg ForbCinPPS_j(T_3, _)
\end{aligned}$$

From the first line of the equation shown above, we can follow $ForbCinPPS_j(T_3, \_)$ which leads to the contradiction and, thus, proves Theorem 1:

$$\Rightarrow \exists j : ForbCinPPS_j(T_3, \_) \wedge \neg ForbCinPPS_j(T_3, \_) \Rightarrow \lightning$$

□

8 Related Work

The *ACTA* framework allows to formalize properties of transaction models using first-order formulas over schedules [6]. Its conciseness and clarity inspired us to implement schedulers based on declarative protocol specifications. The basic ideas of Oshiya have been presented in [13], but this work focused on single-version protocols (2PL) and did not consider correctness. Recent research projects leverage the advantages of declarative languages in various areas [2,4,7,12,15,16]. The *Boom* approach uses Overlog to build distributed systems [2], e.g., a scheduler for MapReduce tasks with policies like First-Come-First-Served. In contrast to our approach, Boom does not focus on DB requests or consistency.

Application analysis techniques have been presented in [10,9] to determine if applications generate serializable executions when running on a system that

applies SI. The key idea is that DBAs analyze transaction programs, produce static dependency graphs and manually check for dangerous access patterns leading to non-serializability. Some approaches modify transaction programs to ensure serializable SI schedules: Fekete [9] proposed the techniques *Materialize* and *Promotion* to achieve serializability. Jorwekar et al. [11] tried to automate the check whether non-serializable SI executions can occur. However, this approach still requires manual confirmation and modification. Fekete [8] executes certain transactions of pivot structures under S2PL, others run under SI. This approach requires the underlying platform to support both S2PL and SI. Alomari et al. [1] set exclusive locks in an *External Lock Manager* (ELM) to ensure serializability with SI. In contrast to DSSI, these approaches do not work for ad-hoc transactions and require static analysis or manual program modifications.

Another line of work focused on modifying the SI algorithm of the underlying system to ensure serializability. The closest approach to DSSI is the SSI protocol [5] described in Sec. 2.3. This approach modifies the DB lock manager with an additional type of locks that are used to detect potential pivot structures. DSSI infers all necessary information to detect and prevent these structures from relation $\mathcal{H}$. Our implementation works with DBMSs out of the box. The underlying DBMS does not even need to provide SI since we model data versions in a standard relational schema (see Sec. 4). Using Oshiya, the implementation of DSSI is close to its formal specification, which enabled us to prove its correctness.

9 Conclusions and Future Work

We develop Declarative Serializable Snapshot Isolation (DSSI) using our declarative scheduling model Oshiya. DSSI ensures serializable schedules by avoiding pivot structures and provides DB independence. We formally define DSSI as an Oshiya protocol specification, present a scheduler implementation, and prove that the implementation ensures serializability. In future work, we will experimentally evaluate the performance of DSSI and investigate the trade-offs involved in reducing the amount of false positives.

References

1. Alomari, M., Fekete, A., Röhm, U.: A Robust Technique to Ensure Serializable Executions with Snapshot Isolation DBMS. In: ICDE, pp. 341–352 (2009)
2. Alvaro, P., Condie, T., Conway, N., Elmeleegy, K., Hellerstein, J.M., Sears, R.: Boom Analytics: Exploring Data-Centric, Declarative Programming for the Cloud. In: EuroSys, pp. 223–236 (2010)
3. Berenson, H., Bernstein, P., Gray, J., Melton, J., O'Neil, E., O'Neil, P.: A Critique of ANSI SQL Isolation Levels. In: SIGMOD, pp. 1–10 (1995)
4. Böhm, A., Marth, E., Kanne, C.-C.: The Demaq System: Declarative Development of Distributed Applications. In: SIGMOD, pp. 1311–1314 (2008)
5. Cahill, M.J., Röhm, U., Fekete, A.D.: Serializable Isolation for Snapshot Databases. TODS 34(4), 1–42 (2009)

6. Chrysanthis, P.K., Ramamritham, K.: ACTA: A Framework for Specifying and Reasoning about Transaction Structure and Behavior. In: SIGMOD, pp. 194–203 (1990)
7. Chu, D., Popa, L., Tavakoli, A., Hellerstein, J.M., Levis, P., Shenker, S., Stoica, I.: The Design and Implementation of a Declarative Sensor Network System. In: SenSys, pp. 175–188 (2007)
8. Fekete, A.: Allocating Isolation Levels to Transactions. In: PODS, pp. 206–215 (2005)
9. 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)
10. Fekete, A.D.: Serializability and Snapshot Isolation. In: Australasian Database Conference, pp. 201–210 (1999)
11. Jorwekar, S., Fekete, A., Ramamritham, K., Sudarshan, S.: Automating the Detection of Snapshot Isolation Anomalies. In: VLDB, pp. 1263–1274 (2007)
12. Kot, L., Gupta, N., Roy, S., Gehrke, J., Koch, C.: Beyond Isolation: Research Opportunities in Declarative Data-Driven Coordination. SIGMOD Rec. 39, 27–32 (2010)
13. Tilgner, C.: Declarative Scheduling in Highly Scalable Systems. In: EDBT/ICDT Workshops, pp. 41:1–41:6 (2010)
14. Weikum, G., Vossen, G.: Transactional Information Systems. Morgan Kaufmann Publishers, San Francisco (2002)
15. White, W., Demers, A., Koch, C., Gehrke, J., Rajagopalan, R.: Scaling Games to Epic Proportions. In: SIGMOD, pp. 31–42 (2007)
16. Yang, F., Shanmugasundaram, J., Riedewald, M., Gehrke, J.: Hilda: A High-Level Language for Data-Driven Web Applications. In: ICDE (2006)