O|R|P|E - A Data Semantics Driven Concurrency Control Mechanism

Tim Lessner*, Fritz Laux†, Thomas M Connolly‡
*freiheit.com technologies gmbh, Hamburg, Germany
Email: tim.lessner@freiheit.com
†Reutlingen University, Reutlingen,Germany
Email: fritz.laux@reutlingen-university.de
‡University of the West of Scotland, Paisley, UK
Email: thomas.connolly@uws.ac.uk

***Abstract*—This paper presents a concurrency control mechanism that does not follow a 'one concurrency control mechanism fits all needs' strategy. With the presented mechanism a transaction runs under several concurrency control mechanisms and the appropriate one is chosen based on the accessed data. For this purpose, the data is divided into four classes based on its access type and usage (semantics). Class O (the optimistic class) implements a first-committer-wins strategy, class R (the reconciliation class) implements a first-n-committers-win strategy, class P (the pessimistic class) implements a first-reader-wins strategy, and class E (the escrow class) implements a first-n-readers-win strategy. Accordingly, the model is called O|R|P|E. Under this model the TPC-C benchmark outperforms other CC mechanisms like optimistic Snapshot Isolation.**

Keywords–Multimodel concurrency control; transaction processing; optimistic concurrency control; snapshot isolation; performance analysis.

I. INTRODUCTION

The drawbacks of existing concurrency control (CC) mechanisms are that pessimistic concurrency control (PCC) is likely to block transactions and is prone to deadlocks, optimistic concurrency control (OCC) may experience a sudden decrease in the commit rate if contention increases. Snapshot Isolation (SI) better supports query processing since transactions generally operate on snapshots and also prevents read anomalies, but depending on the implementation of SI, either pessimistic or optimistic, it is also subject to the previously mentioned drawbacks of PCC or OCC. Semantics based CC (SCC) such as the mechanism proposed in [1] remedies some problems of PCC or OCC. It performs well under contention, reduces the blocking time, and better supports disconnected operations. However, its applicability is limited since data and transactions have to comply with specific properties such as the commutativity of operations. In addition to the previously mentioned drawbacks, neither PCC nor OCC nor SCC support long-lived and disconnected data processing. However, these properties are essential to achieve scalability.

This paper presents a mechanism originally introduced in [2] that combines OCC, PCC, and SCC and steps away from the **'one concurrency control mechanism fits all needs'** strategy. Instead, the CC mechanism is chosen depending on the data a transaction accesses. To address scalability, the mechanism was designed with a focus on long-lived and disconnected data processing.

Consider, for example, the whole sales scenario of the TPC-C [3]. With PCC using shared and exclusive locks, the likelihood of deadlocks increases for hotspot fields such as the stock's quantity or the account's debit or credit. If transactions are long-lived, PCC is even worse since deadlocks manifest during write time and a significant amount of work is likely to be lost [4], [2]. With OCC, deadlocks cannot occur. However, hot-spot fields like an account's debit or credit would experience many version validation failures under high load causing the restart of a transaction. Like PCC, validation failures manifest during the write-phase of a transaction and a significant amount of work is likely to be lost. Both PCC and OCC cannot ensure that modifications attempted during a transaction's read-phase will prevail during the write-phase. Wheras PCC is prone to deadlocks (in the case of shared locks), OCC is prone to its optimistic nature itself.

O|R|P|E resolves these drawbacks and data can be classified in CC classes. For example, customer data such as the address or password can be controlled by a PCC that uses exclusive locks only and performs lock pre-claiming [5]. Such a rigorous measure ensures ownership of data and should be used if data is modified that belongs to one transaction. For example, account data or master data should not be modified concurrently and given the importance of this data a rigorous isolation is justififed. The debit or credit of an account can be classified in CC class R, which guarantees no lost updates and no constraint violations. Such a guarantee is often sufficient for hot-spot fields. Class E can be used to access an item's stock, for example. Class E is able to handle use cases such as reservations. It should be used if during the read-phase a guarantee is required that changes will succeed during the write-phase. Class O is the default class. It avoids blocking and under normal load it represents a good trade-off between commit and abort-rate.

Section II defines these four CC classes with different data access strategies used by the mechanism. In case of a conflict, class O implements a **first-committer-wins** strategy, class R implements a **first-n-committers-win** strategy, class P implements a **first-reader-wins** strategy, and class E implements a **first-n-readers-win** strategy. The number n is determined by the semantics of the accessed data, e.g., by database constraints. According to the classes, the mechanism is called O|R|P|E. The "|" indicates the demarcation between data.

 ISBN: 978-1-61208-408-4

Section III proofs the correctnes of the model. Section IV briefly describes the prototype implementation. Section V highlights some advantages of O|R|P|E, because it provides an application flexibility in choosing the best suitable CC mechanism and thereby significantly increases the commit rate and outperforms optimistic SI. Finally, the paper summarizes related work (see Section VI) and provides an outlook (see VII).

II. MODEL

A. Transaction

To support long-lived and disconnected data processing, which both supports scalability, O|R|P|E models a transaction as a disconnected transaction τ, with separate read- and write-phase, i.e., no further read after the first write operation (see Definition II.1, taken from [2]). To disallow blind writes O|R|P|E guarantees that in addition to the value of a field, the version of a data field has to be read, too.

DEFINITION II.1: Disconnected Transaction:

1) Let ta be a flat transaction that is defined as a pair $ta = (OP, <)$ where OP is a finite set of steps of the form $r(x)$ or $w(x)$ and $<(\subseteq OP \times OP)$ is a partial order.
2) A disconnected transaction $\tau = (TA^R, TA^W)$ consists of two disjoint sets of transactions. $TA^R = \{ta_1^R, \ldots, ta_i^R\}$ to read and $TA^W = \{ta_1^W, \ldots, ta_j^W\}$ to write the proposed modifications back.
3) A transaction has to read any data item x before being allowed to modify x (no blind writes).
4) If a transaction only reads data it has to be labelled as read only.

B. CC Classes

Class O is the default class and is implemented by an optimistic SI mechanism, which is advantageous since reads do not block writes and non-repeatable or phantom phenomena do not happen. However, SI is not serializable [6], [7].

As stated, the drawback of optimistic mechanisms prevails if load increases, because many transactions may abort during their validation at commit time. An abort at commit time is expensive, because significant amount of work might be lost. A circumstance particularly crucial for long-lived transactions (see [2]).

Regarding the strategy, optimistic SI follows a "first-committer-wins" semantics revealing another drawback of O. It is the lack of an option allowing a transaction to explicitly run as an owner of some data. Consider, for example, the private data of a user such as its password or address. A validation failure should be prevented by all means, since it would mean that at least two transactions try to concurrently update private data. Although technically this is a reasonable state, for this kind of data a pessimistic approach that acquires all locks at read time is more appropriate. Such a mechanism follows a "first-reader-wins" (ownership) semantics and directly leads to class P. Lock acquisition at read time enables a strict sequential access and preclaiming (all reads and locks appear before the first write) prevents deadlocks during the write-phase if exclusive locks are used.

The decision if a data item is classified as O or P is based on the following properties [2]:

1) *Mostly read* (mr): Is the data item mostly read? If 'Yes', there is no need for restrictive measures and the data item should by classified for optimistic validation. A low conflict probability is assumed.
2) *Frequently written* (fw): fw is the opposite of mr.
3) *unknown* (un): It means neither mr nor fw apply, i.e., it is unknown whether an item is mostly read or written or approximately even.
4) *Ownership* (ow): if accessing a data item should explicitly cause the transaction to own this item for its lifetime?

EXAMPLE II.1: Classify data items in class O and P (taken from [2]).

This example is based on the TPC-C [3] benchmark and its "New-Order" transaction. Note that an additional table Account has been introduced to keep track about a customer's bookings (column debit and credit). It also defines an overdraft limit (column limit). The following tables are used in our example: Customer (id, name, surname), Stock (StockId, ItemId, quantity), Account (AcctNo, debit, credit, limit), and Item (ItemId, name, unit, price). Table I shows an initial classification.

Attributes *name*, *surname*, and *id* of a customer are expected to be mostly read, but if modified by a transaction it should definitively be the owner. The *id* of a customer, like all ids, is expected to become modified rarely. If the *id* becomes modified, ownership is required. In principal, all business keys should be classified in P, because they are owned by the application provider (see Rule II.1 (1)).

Stock.quantity is expected to become modified frequently (fw) and to prevent the situation where an item was marked as available during the read phase, but at commit time the item is no longer available due to concurrent transactions, it is also marked as ow. For the time being, however, *quantity* will be classified as an ambiguity (see also Rule II.1 (3)), which will be discussed below.

The *Account.credit* and *Account.debit* of a customer's account might be accessed frequently depending on a customer's activity and un is a good choice. However, since multiple transactions might concurrently update the balance, and an owner is hardly identifiable, $\neg ow$ is chosen. So, it is also an ambiguity (see Rule II.1 (3)).

The *Account.limit* is the overdraft limit of a customer and expected to be mostly read, hence, mr is a good choice. Since it is neither owned by the customer nor by others, $\neg ow$ is a good choice (see Rule II.1 (2)).

Assuming the application is a high frequency trading application, *Item.Price* might quickly become a bottleneck. An exact prediction is not possible though, hence, un is a good choice. Property ow would not be a good choice, because transactions of different components (dc) might simultaneously calculate the price (see Rule II.1 (3)).

The ambiguities A of Example II.1, see class A in Table I, highlight that classes O and P and their properties are not sufficient. Particularly, hot spot items such as *Stock.quantity* would benefit from a CC mechanism that allows many winners and resolves the drawbacks of OCC and PCC.

 ISBN: 978-1-61208-408-4

TABLE I. CLASSIFICATION OF EXAMPLE II.1

x	mr	fw	un	ow	CC class
Customer.name	1	0	0	1	P
Customer.surname	1	0	0	1	P
Customer.id	1	0	0	1	P
Stock.StockId	1	0	0	1	P
Stock.ItemId	1	0	0	1	P
Stock.quantity	0	1	0	1	**A**
Account.debit	0	0	1	0	**A**
Account.credit	0	0	1	0	**A**
Account.limit	0	0	1	0	**A**
Item.name	1	0	0	1	P
Item.unit	1	0	0	1	P
Item.price	0	0	1	0	**A**

Laux and Lessner [1] propose the usage of a mechanism that reconciles conflicts –class R–. Their approach is an optimistic variant of O'Neil's [8] Transactional Escrow Method (TEM). Both approaches exploit the commutativity of write operations. If operations commute, it is irrelevant which operation is applied first as long as the final state can be calculated (see [1], [2] for further details) and no constraint is violated.

Unlike TEM, the reconciliation mechanism requires a dependency function. Consider, for example, two transactions that update an account and both read an initial amount of 10€, one credits in 20€ and the other debits 10€. Once both have committed, it is relevant that no constraint was violated at any time and the final amount has to be 20€. Usually, a database would write the new state for each transaction causing a lost update. A dependency function would actually add or subtract the amount (the delta!) and would always take the latest state as input. In other words, reconciliation replays the operation in case of a conflict. However, this is only possible if no further user input is required. In the example above this means the user wants to credit 10€ (or debit 20 €) independent of the account's amount as long as no constraint is violated! Another requirement is that each dependency function has to be compensatable (see also [2]).

The reconciliation mechanism [1] follows a "first-n-committers-win" semantics and the number of winners n is solely determined by constraints. The correctness of the mechanism is proven in [1] which also introduces "Escrow Serializability", a notion for semantic correctness.

TEM grants guarantees to transactions during their read-phase. For example, a reservation system is able to grant guarantees to a transaction about the desired number of tickets as long as tickets are available. The consequence is that transactions need to know their desired update in advance (see [8] for further details).

Whereas TEM [8] is pessimistic (constraint validation during the read phase) and works for numerical data only, Reconciliation [1] is optimistic (constraint validation during the write phase) and works for any data as long as a dependency function is known. The proof that E, like R, is escrow serializable can be found in [2].

The decision if an item is member of R or E is based on the following properties:

1) con: Does a constraint exist for this data item?
2) num: Is the type of the data item numeric?
3) com: Are operations on this data item commutative?
4) dep: Is a dependency function known for an operation modifying the data item?
5) in: Is user input independence given for an operation modifying the data item?
6) gua: Is a guarantee needed that a proposed modification will succeed?

TABLE II. ILLUSTRATIVE CLASSIFICATION OF AMBIGUITIES OF EXAMPLE II.1.

x	con	com	num	dep	in	gua	CC class
Stock.quantity	1	1	1	1	0	1	E
Account.credit	1	1	1	1	1	0	R
Account.debit	1	1	1	1	1	0	R
Item.price	0	0	1	1	0	0	O

RULE II.1: Derivation of CC classes for data item x

1) $ow \rightarrow$ classify x in P (identify P).
2) $\neg ow \wedge mr \rightarrow$ classify x in O (identify O).
3) all other combinations of ow and mr: classify x in A (ambiguity).
4) $com \rightarrow$ classify x in $E \cup R$
   a) $(con \wedge num \wedge com \wedge gua) \rightarrow$ classify x in E (identify E).
   b) $(in \wedge dep \wedge com) \rightarrow$ classify $x \in R$ (identify R).
5) $x \in A \rightarrow$ item x will be eventually in O.

EXAMPLE II.2 (Classification of data items in R and E)**:** The ambiguities of Table I are the input for this example. Table II shows the result of the classification of these ambiguities.

`Stock.quantity` has a constraint $value > 0$ and is numeric. The dependency function dep is known too. As stated above, a dependency function performs a context dependent write. For example, dependency function d would be $d(x, xread, xnew) = x + (xnew - xread)$. User input independence in is not given. If placing the order fails at the end, a replay would also fail. So, class R is not an option. Since an order requires a guarantee that the requested amount of items remains available, Rule II.1 (4 a)) applies.

`Account.credit` and `Account.debit` are classified as R. Property dep is known, because operations are either additions or subtractions. Property in is given, because the account has to be updated if the order is placed and no constraint is violated. As the updates follow a dependency functions they can be reconciled and should not raise an exception. Again, only a constraint violation such as an overdraft can cause the abort. Rule II.1 (4 b)) applies.

`Item.price` depends on a variety of parameters including the last price itself. As a result, a price update might not be commutative. `Item.price` remains ambiguous and remains in O, because O is the default class. Rule II.1 (5) applies

III. CORRECTNESS

A transaction potentially runs under four different CC mechanisms. Due to the CC classes' individual semantics, each class has a different notion for a conflict, too.

Usually, a conflict is given if two operations access the same data item and the corresponding transaction overlap in their execution time, and at least one operation writes the data item [5]. Whereas for O this is a correct definition of a

 ISBN: 978-1-61208-408-4

conflict, for R and E it is not, because both can resolve certain write conflicts. The resolution of conflicts is a key aspect and advantage of SCC, and SCC questions the seriousness of a conflict. In other words the meaning of a read-write or write-write conflict is a interpreted. For R and E only a constraint violation is a conflict. Moreover, the state read by an operation is assumed to be irrelevant, otherwise commutativity is not given. It follows that any final serialization graph $SG-R$ and $SG-E$ for class R and E is non-cyclic because potential conflicts are reconciled (see [2] for a thorough discussion).

For P, the common definition of a conflict is correct, but the peculiarities of lock preclaiming during the read phase mean that if a transaction wants to modify item p (let $p \in P$), it has to acquire a lock on p during its read-phase to become the exclusive owner. If not, the transaction does a blind write, which is disallowed according to Definition II.1. Hence, every write in P cannot encounter a concurrent write or read, because if a transaction writes p it has to be the exclusive owner of P.

Lock preclaiming in P takes place at the begin of a read phase. Consider the following schedule, for example (let $disc_i$ denotes the disconnect phase of transaction i and let $o \in O$ and $p \in P$):

$$r_i(o), r_j(p), r_j(o), disc_j, w_j(p), c_j, r_i(p), disc_i, w_i(p), c_i$$

Transaction i precedes j in class O and j precedes i in P. Having contradicting orders, i.e., $i \rightarrow j$ in one, but $j \rightarrow i$ in another class violates serializability. If lock preclaiming is the first step carried out during a transaction's read phase, this unfavourable situation between classes O and P is avoided. Consider the following schedule, for example:

$$r_i(p), r_i(o), r_j(p), r_j(o), disc_i, w_i(p), c_i, disc_j, w_j(p), c_j$$

Now, transaction j has to wait until transaction i has committed, because j cannot be owner of p if i owns p.

Based on these findings it is possible to state Theorem III.1. The corresponding proof III.1 exploits that for R, P, and E the corresponding serialization graphs are non-cyclic.

THEOREM III.1: Let $SG-G$ be the global serialization graph, which is the union of $SG-O$, $SG-R$, $SG-P$, and $SG-E$. The global serialization graph $SG-G$ is non-cyclic if $SG-O$ is non-cyclic.

PROOF III.1 (By contradiction)**:** Given that ta_i is serialized before ta_j $(i \rightarrow j)$ in $SG-O$. In P, no other transaction can access an item in P if transaction ta_i has read the item. This includes ta_j and it is impossible to have a serialization order $j \rightarrow i$ in P. Since i and j can be arbitrarily changed there is a contradiction if $i \rightarrow j$ exists in one, and $j \rightarrow i$ in another class. $SG-R$ and $SG-E$ are negligible because any conflict is finally reconciled and both serialization graphs are non-cyclic.

COROLLARY III.1: $SG-O$ sets the global serialization order for P.

If a ta does not modify data in O, then P sets the order. If a ta does not modify data in P, then R sets the order, because it is prone to validation conflicts as opposed to E that already has a guarantee to succeed.

IV. PROTOTYPE REFERENCE IMPLEMENTATION OF O|R|P|E

The prototype of O|R|P|E is not a full database system. It was implemented using the JAVA programming language and Figure 1 illustrates its architecture. A client API provides access to the data and depending on the operation's type read or write, the operation is executed by a dedicated pool. Pools' "reads" and "writes" represent an read- and write-lane. In addition, a pool to handle the termination (commit and abort) has been implemented. Pools' reads and writes handle all incoming and outgoing operations and the classification has been placed directly into the index. Depending on an item's classification the corresponding CC mechanism is plugged in. Once an item has been read or written, the additional pools' "read-callback" and "write-callback" deliver the results back to the clients. Pool WFG (Wait-for-Graph) is used to handle access to the WFG. Deadlocks may occur during the read-phase of a transcation if the transcation accesses data items in class P. Deadlocks can occur in class P during the read-phase, because lock acquisition is not globally ordered.

Having separate pools to handle incoming and outgoing operations means that the prototype supports disconnected transactins, because the entire communication is asynchronous. There is no single thread that represents a specific client or transaction. Figure 2 illustrates the message flow within the prototype. A read operation is passed to the "reads" pool. Each read is executed asynchronously and the complete read set is sent back to the client via a dedicated callback pool. To support asynchronous writes, a write operation is passed to the "writes" pool and if all writes have been applied the write set is sent back to the client. Clients always sent their complete write-set.

Data is kept solely in memory and no data is written to disk unless the operating system needs to swap data to disk due to memory limitations. The only output to disk is to write logging events that are used for performance evaluation. Other functionality that has been implemented includes:

- CC mechanisms O, R, P and E as well as P;
- The prototype supports constraints.
- The prototype supports selects, range-selects, updates, and inserts. The deletion of an item is an update that invalidates a data item.
- A WFG implementation.

V. PERFORMANCE STUDY

The performance study has been carried out based on the prototype presented in the previous section (Section IV). As benchmark, the TPC-C++ benchmark [7] has been chosen, because we also conducted a study comparing O|R|P|E with Serializable SI, which is beyond the scope of this paper.

The performance study measures the response-time (resp. -time), the abort rate (ab-rate), the degree of concurrency (deg. conc.), and commits per second. The degree is the quotient of the serial estimated time over the elapsed time of the experiment. In addition, the arrival rate λ of new transactions has been varied to be set to the optimum (minimised abort rate and response time, maximised degree of concurrency). This optimum λ has been taken to conduct fair and calibrated comparisons. Each experiment has been repeated three times and the mean value is reported. Values refer to the execution

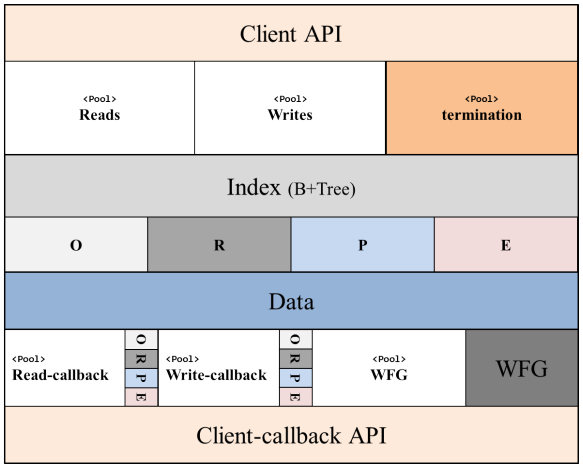

Figure 1. Architecture of the prototype.

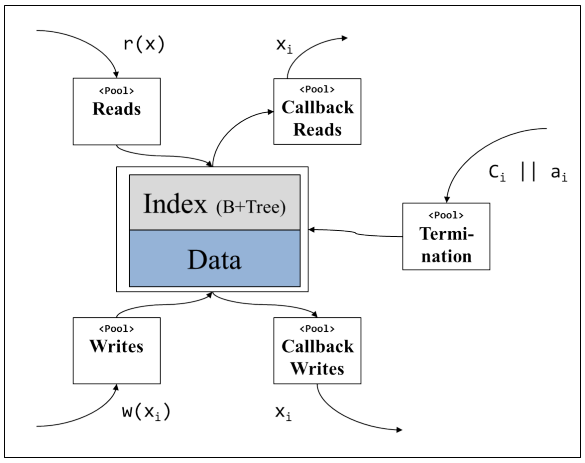

Figure 2. Message flow of the prototype.

of a transaction mix –deck– (42 New Order-, 42 Payment-, 4 Delivery-, 4 Credit check-, 4 Update Stock Level-, and 4 Read Stock Level - transactions see [7], [3], [2]). The classification of data shown in Table III is similar to the classification of Examples II.1 and II.2.

Figure 3 illustrates the abort rate and degree of concurrency for SI under full contention and shows the drawbacks of optimistic SI: the higher the number of concurrent transactions, the higher the abort rate. Also, the system starts thrashing if

TABLE III. TPC-C: CLASSIFICATION OF DATA ITEMS.

Item	CC Class	operation
Customer	P	read
CustomerCredit	P	update
CustomerBalance	R	read
Customer	P	read
CustomerBalance	R	update
Customer	P	read
CustomerCredit	P	read
StockQuantity	E	update
Customer	P	read
CustomerBalance	R	update
WarehouseYTD	R	update
DistrictYTD	R	update
StockQuantity	E	read only
StockQuantity	E	update

the degree of concurrency drops below one, which is the point where a serial execution outperforms a concurrent. Table IV #1-6 shows that the response-time increases with larger λ, which is expected and normal behaviour. A good degree of concurrency with a low abort rate is given by $\lambda = 133$ (see Table IV #3).

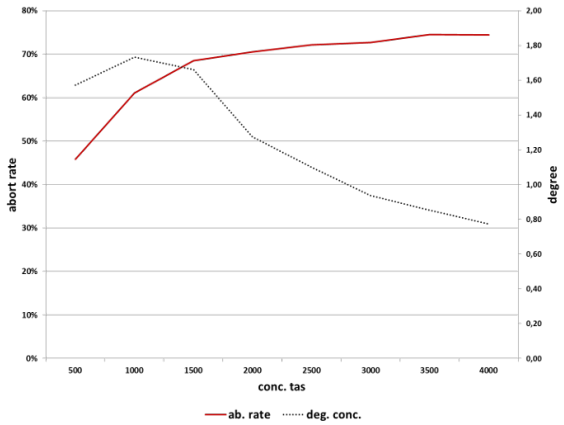

Figure 3. TPC-C++, optimistic SI (class O), abort rate and degree of concurrency.

Figure 4 shows the response-time and degree of concurrency for O|R|P|E for increasing λ. Unlike SI, O|R|P|E has no aborts caused by serialization or validation conflicts due to the classification of hot-spot data items in R or E, which prevents ww-conflicts. As shown by Figure 4, O|R|P|E has its best degree with $\lambda = 1000$ transactions per second achieving 227 commits per second (see Table IV, #15).

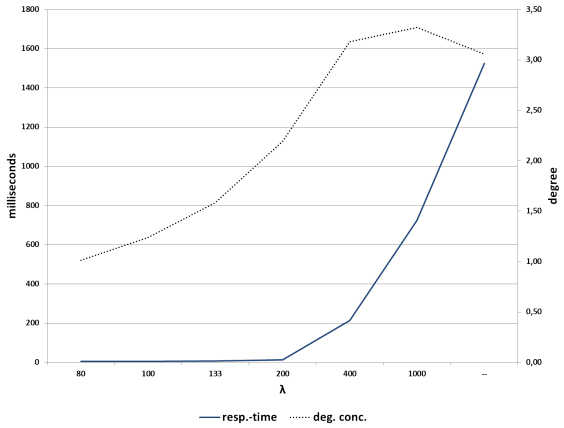

Figure 4. Response time and degree of concurrency for increasing λ for O|R|P|E .

The comparison of O|R|P|E and SI uses $\lambda = 133$ (Table IV #3, and #7-9) for SI and $\lambda = 1000$ (Table IV #15-18) for O|R|P|E . For SI, $\lambda = 133$ was considered as being the best trade-off with respect to the degree of concurrency, $\lambda = 1000$ was considered as being the best trade-off for O|R|P|E. Figure 5 illustrates the degree and the response-time for O with SI and O|R|P|E if both use the λ which reflect the best trade-off. As the figure shows, SI has a better response-time for 1000, 2000, and 3000 concurrent transactions, but then suddenly undergoes thrashing and the response-time grows exponentially. However,

 ISBN: 978-1-61208-408-4

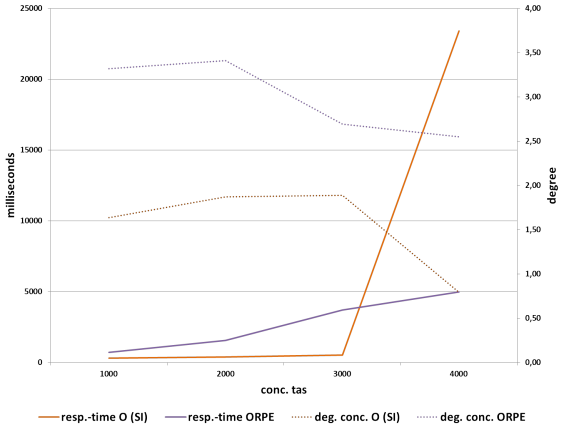

Figure 5. TPC-C++, SI and O|R|P|E : response-time and degree of concurrency for $\lambda = 133$ (SI) and $\lambda = 1000$ (O|R|P|E).

TABLE IV. MEASURED VALUES OF ALL EXPERIMENTS #1-18.

#	tas	λ	resp.-time	ab. rate	commits /second	deg. conc.
1	1000	80	43	2%	71	1,39
2	1000	100	84	3%	80	1,57
3	1000	133	309	5%	82	1,63
4	1000	200	1640	20%	62	1,50
5	1000	400	2091	26%	61	1,57
6	1000	1000	2464	27%	62	1,61
7	2000	133	388	9%	90	1,87
8	3000	133	522	8%	91	1,89
9	4000	133	23416	46%	22	0,79
10	1000	80	5	4%	69	1,01
11	1000	100	5	4%	85	1,24
12	1000	133	8	4%	108	1,58
13	1000	200	14	4%	150	2,19
14	1000	400	213	4%	217	3,18
15	1000	1000	724	4%	227	3,32
16	2000	1000	1551	4%	234	3,41
17	3000	1000	3704	4%	184	2,69
18	4000	1000	4968	5%	174	2,55

O|R|P|E shows a moderate and stable increase of the response-time even for 4000 concurrent transactions. It would be wrong to conclude that SI has a better performance than O|R|P|E for workloads below 4000 concurrent transactions, because λ has to be taken into account and for O|R|P|E $\lambda = 1000$ as opposed to $\lambda = 133$ for SI. Hence, under high contention O|R|P|E has the lowest abort rate and considering the trade-off, O|R|P|E has the shortest response-time. Furthermore, the abort rate is independent of the contention.

The drawback of O|R|P|E is that a wrong classification of data can quickly cause performance issues. For example, P as well as E are expensive, because P requires locking during the read-phase and E has to validate all constraints against each other. There is also overhead to classify the data. If a classification is not possible, class O is the default class –there is a fallback–.

VI. RELATED WORK

This paper is based on the findings of [2], which introduces O|R|P|E. A vast amount of work [5], [9] has been carried out in the field of transaction management and CC, but so far no attempt was undertaken to use a combination of mechanisms according to the data usage (semantics). Most authors use the semantics to divide a transaction into sub-transactions thus achieving a finer granularity that hopefully exhibit less conflicts. Some authors [10] use the semantics of the data to build a compatibility set while others try to reduce conflicts using multiversions [11], [12]. The reconciliation mechanism was introduced by [1] and is an optimistic variant of [8] "Transactional Escrow Method". Escrow relies on guaranties given to the transaction before the commit was executed, which is only possible for a certain class of transactions. OCC was introduced by [13], which did not gain much consideration in practice until SI, introduced by [14], has been implemented in an optimistic way. SI in general gained much attention through [6], [7], also in practice [15].

VII. OUTLOOK

The work carried out in [2] covers, in addition to the presented result, various aspects of O|R|P|E such as replication and runtime adaptation. However, a dynamic algorithm for an automatic classification of data would be advantageous. Also, a performance study that considers replication and runtime adaptation is still missing.

REFERENCES

[1] F. Laux and T. Lessner, "Transaction processing in mobile computing using semantic properties," in *Proceedings of the 2009 First International Conference on Advances in Databases, Knowledge, and Data Applications*, ser. DBKDA '09. IEEE Computer Society, 2009, pp. 87–94.

[2] T. Lessner, "-ORPE- a high performance semantic transaction model for disconnected systems," Ph.D. dissertation, University of the West of Scotland, 2014.

[3] *TPC BENCHMARK C, Standard Specification, Revision 5.11*, Transaction Processing Performance Council Std., February 2010.

[4] A. Thomasian, "Concurrency control: methods, performance, and analysis," *ACM Comput. Surv.*, vol. 30, no. 1, pp. 70–119, Mar. 1998.

[5] J. Gray and A. Reuter, *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[6] A. Fekete, D. Liarokapis, E. O'Neil, P. O'Neil, and D. Shasha, "Making snapshot isolation serializable," *ACM Trans. Database Syst.*, vol. 30, no. 2, pp. 492–528, Jun. 2005.

[7] M. J. Cahill, U. Röhm, and A. D. Fekete, "Serializable isolation for snapshot databases," *ACM Trans. Database Syst.*, vol. 34, no. 4, pp. 20:1–20:42, Dec. 2009.

[8] P. E. O'Neil, "The escrow transactional method," *ACM Transactions On Database Systems*, vol. 11, pp. 405–430, December 1986.

[9] G. Weikum and G. Vossen, *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. Morgan Kaufmann, 2002.

[10] H. Garcia-Molina, "Using semantic knowledge for transaction processing in a distributed database," *ACM Trans. Database Syst.*, vol. 8, no. 2, pp. 186–213, Jun. 1983.

[11] S. H. Phatak and B. Nath, "Transaction-centric reconciliation in disconnected client-server databases," *Mob. Netw. Appl.*, vol. 9, no. 5, pp. 459–471, 2004.

[12] P. Graham and K. Barker, "Effective optimistic concurrency control in multiversion object bases," in *ISOOMS '94: Proceedings of the International Symposium on Object-Oriented Methodologies and Systems*, ser. Lecture Notes in Computer Science, E. Bertino and S. D. Urban, Eds., vol. 858. London, UK: Springer-Verlag, 1994, pp. 313–328.

[13] H. T. Kung and J. T. Robinson, "On optimistic methods for concurrency control," *ACM Trans. Database Syst.*, vol. 6, no. 2, pp. 213–226, Jun. 1981.

[14] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil, "A critique of ansi sql isolation levels," *SIGMOD Rec.*, vol. 24, no. 2, pp. 1–10, May 1995.

[15] D. R. K. Ports and K. Grittner, "Serializable snapshot isolation in postgresql," *Proc. VLDB Endow.*, vol. 5, no. 12, pp. 1850–1861, Aug. 2012.

 ISBN: 978-1-61208-408-4