Effective Partial Order Reduction in Model Checking Database Applications

Maryam Abdul Ghafoor Muhammad Suleman Mahmood Junaid Haroon Siddiqui
LUMS School of Science and Engineering, Lahore, Pakistan
Email: {15030036,13030004,junaid.siddiqui}@lums.edu.pk

***Abstract*—Distributed applications, in particular web applications, often depend on a centralized database. The results of database operations depend on the state of database at that time and often also on the order of execution of operations performed by concurrent clients. Verification of such applications requires modeling all these possible orders so that the user can determine which are incorrect orderings and can prevent them with transactions or business logic. However, straightforward exploration leads to state space explosion. Partial order reduction prunes orderings that are equivalent to other orderings already explored.**

We present a novel technique of Effective Partial Order Reduction (EPOR) for model checking software of Java applications sharing database state. EPOR improves upon prior work by performing a more precise analysis and supports many more operations. The key idea behind EPOR is that monitoring the effect of database operations inside database implementation gives a more precise view of operation dependencies than what can be achieved from an external view. Like prior work, EPOR also relies on Java Pathfinder model checker for model checking Java application. However, unlike prior work, there is additional instrumentation inside the database that enables our precise analysis and allows supporting more constructs. Our results improve upon prior work by achieving significant reduction in number of states explored and thus enables more effective model checking of database applications with concurrent operations.

***Index Terms*—Partial Order Reduction, Database Applications, Model Checking**

I. Introduction

With advancement of computer technology, highly concurrent systems are being developed. Relational Database Management Systems (RDBMS) are widely used for storing and managing data for applications. Normally, a single database server serves multiple client applications. RDBMS store data in tables (relations). Multiple clients can access the same tables at the same time. The accesses can be writes to the database tables (inserts, update and delete statements) or reads from the database tables (select statements). Moreover, due to these operations database state is changing continuously. At a given time multiple clients are sharing the same database state. Multiple clients are accessing these tables for different operations and their output or result of access depends on the state of database at that time. Order of execution of database operation depends on the arrival of client request, which could be different for different cases depending on factors like network traffic etc. Many different schedulings (order of processes) of requests are possible in such situations.

One of the major problems in automatic verification of such systems is state space explosion. In a given schedule, if two operations share database state, they are dependent. Result of the database access truly depends on the state of database at that time. Although database semantics inherently ensure atomicity and have strict concurrency control [1], but based on the dependence of two operations, the outcome of database access for one client can be non-deterministic. That can lead to overall unpredictable behavior of an application. Since the number of possible schedules are permutations of the number of active processes, with increase in the number of processes, the number of different schedules grow exponentially. Number of schedules is the most important factor in handling state space explosion for concurrent programs.

Automated testing of such behavior is challenging task. There are several techniques that address automated testing of databases. Software model checking [2]–[4] is one of the techniques that verifies properties of concurrent systems. It deals with exploration of all possible interleaving of processes leading to state space explosion for larger set of processes. In general, concurrent events are modelled by exploring all possible interleaving of events relative to each process which results in a large set of paths with many states on each path. Each path represents one unique interleaving of processes. Model checkers employ partial order reduction to avoid unnecessary exploration of schedules. Partial order reduction addresses the problem of state explosion for concurrent systems by reducing the size of state space to be searched by model checking software. Partial order reduction [5]–[7] on shared variables and objects is a well-researched field. It exploits the commutativity of concurrently executed transitions, which results in the same state when executed in different orders. It consider only a subset of paths representing a restricted set of behaviors of the application while guarantees that ignored behaviors do not add any new information.

Database PathFinder (DPF) [8] introduces model checking of database applications and describes partial-order reduction techniques in the context of database applications. However, they treat the database as an external entity and perform partial order reduction at the SQL language level. This forces them to take some pessimistic decisions about dependencies of queries on each other. Furthermore, working at the SQL level makes it difficult to handle complex queries. We address these problems in Effective Partial Order Reduction (EPOR) with a more precise analysis that further reduces the number of states.

Our key idea is to observe the effect of SQL statements by instrumentation inside the database engine. Instead of analyzing SQL statements, we analyze which rows are read from and written to during execution of a particular SQL statement and if needed, temporarily observe the effect of SQL statements applied in opposite order. Analyzing the effect of SQL statements inside the database reduces the need to parse and process every SQL statement separately and the automatic reexecution based analysis enables much precise partial order reduction than what was proposed in DPF.

We are using PostgreSQL database engine[1] and Java PathFinder (JPF) [9] model checker for our implementation. We consider concurrent Java processes interacting with a database through SQL statements. We model processes as threads in a multi-threaded application and use JPF to generate thread schedules for the application. We find commutative paths by analyzing database accesses and interdependency of PostgreSQL statements executed by multiple threads. The actual queries are executed in database which is run along with the model checker to check record sharedness and to mark dependency among records of same tables. Rows with possible read write conflicts, e.g., when an INSERT query affects a SELECT query only if specific data is inserted are identified by rerunning the SELECT to identify precise dependency. The results obtained are used by our dependency analyzer to mark statements dependent for exploration of other interleavings.

We make the following contributions:

- **More Precise Partial Order Reduction.** Our Effective Partial Order Reduction technique (EPOR) is more precise as it finds dependencies among queries by executing them and observing them inside the database.
- **Instrumentation of PostgreSQL.** We instrumented PostgreSQL query nodes to extract unique row identifiers to perform precise dependency analysis and enable depedency analysis of many SQL operations.
- **Implementation.** We implemented partial order reduction for Java applications using PostgreSQL. Our algorithms for partial order reduction are adaptable to other databases systems as well. Based upon ideas in DPF, we also implement database state restoration with JPF backtracking. We exploit depth first search mechanism of JPF to map JPF state identifiers to database save points for efficiently restoring database state while backtracking through program states.
- **Evaluation.** We evaluated our technique on PetClinic4[2] which is an official sample distributed with Spring framework. We also evaluated our technique in comparison of DPF and observe significant reduction in state space size in some cases.

II. Motivating Example

In order to explain dependent database accesses i.e. dependency of PostgerSQL statements on each other, which form the basis of partial order reduction, we consider a simple example of an Employee Management System. Our database schema for this example is a single table COMPANY which contains records of the employees of the company. It has four attributes named ID, Name, Location, and Salary. We have a primary key constraint on the ID column. To illustrate this example, three rows of table COMPANY are shown in Table Ia. There can be multiple operations that can be performed on this table. In our application, we consider only two operations: `addBonus(int maxSalary, int Bonus)` and `updateSalary(int increment, String loc)`. Function addBounus basically adds bonus amount to the employee's salary if salary is less than given threshold value maxSalary. So inputs for this function are bonus and maxSalary. The other function, updateSalary increments the salary of the employees based on the location. Location(loc) and increment are two input parameters for this operation. We have mapped these two operations on two threads such that each operation is performed by a different thread. Code snippet for both operations, executed by different threads, is shown in Fig. 1 and Fig. 2.

```
1  void addBonus(int maxSalary, int Bonus){
2    int id = 0;
3    int newSalary = 0;
4    int bonus = Bonus;
5    Connection conn = DriverManager.getConnection
         (url, "postgres","");
6    Statement stmt = conn.createStatement();
7    ResultSet rs = stmt.executeQuery("SELECT ID,
         Salary FROM Company WHERE
         Salary<maxSalary");
8    if (rs.next()) {
9      id = rs.getInt("ID");
10     newSalary = rs.getInt("Salary");
11     newSalary = newSalary+ bonus;
12     stmt.executeUpdate("UPDATE Company SET
           Salary="+newSalary+" WHERE ID="+id);
13     newSalary = 0;
14   }
15   stmt.close();
16   rs.close();
17   conn.close();
18 }
```

Fig. 1: Example Code for identification of data dependence problem when executed by two processes/ threads. It adds bonus to Employees Salary, if their salary is less than maxSalary, passed as input parameter

Consider ThreadSchedule-I given in Fig. 3. The SQL statements accessing the database by Thread 1 for addBonus operations are statements 7 and 12. Statement 7 is retrieving IDs of those employees who have salary less than maxSalary passed as input to the function and then on line 12, there is SQL statement for modifying the salary of previously retrieved employees by adding bonus to it. Thread 2 is performing updateSalary operation and it accesses the database through statement 4 where based on the location, salary of employees is incremented by value passed to this function as integer.

[1] http://www.postgresql.org

[2] http://static.springsource.org/docs/petclinic.html

```
1  void updateSalary(int increment, String loc){
2    Connection conn =
         DriverManager.getConnection(url,
         "postgres","");
3    Statement stmt = connection.createStatement();
4    stmt.executeUpdate("UPDATE Company SET
         Salary=Salary+"+increment+" WHERE
         Location="+loc);
5    stmt.close();
6    conn.close();
7  }
```

Fig. 2: Example Code for identification of data dependence when executed by two processes/ threads. It updates the salary of the employees based on their location

```
Thread 1: Line 7   addBouns: SELECT
Thread 1: Line 12  addBonus: UPDATE
Thread 2: Line 4   updateSalary: UPDATE
```

Fig. 3: ThreadSchedule-I (Thread 1 → Thread 2)

```
Thread 2: Line 4   updateSalary: UPDATE
Thread 1: Line 7   addBouns: SELECT
Thread 1: Line 12  addBonus: UPDATE
```

Fig. 4: ThreadSchedule-II (Thread 2 → Thread 1)

When ThreadSchedule-I is executed with test inputs (input 1) [maxSalary=12000, Bonus=500] and [increment=2000, loc='Texas'] for addBonus and updateSalary respectively. Thread 1 selects an employee of table COMPANY with ID 2 and modifies its salary by adding 500 to it. After completion of addBonus operation, Thread 2 performs updateSalary operation based on the location. This time Thread 2 will modify the salary of employee with ID 2 changing the state of database again. Final database state after the completion of ThreadSchedule-I, is given as in Table Ib.

In the other schedule, given in Fig. 4, Thread 2 performs updateSalary operation followed by addBonus operation by Thread 1. Given the initial state of database in Table Ia and same input as before, Thread 2 updates the salary of employee with ID 2 to 12500. Then Thread 1 executes addBonus. After the execution of the both operations for ThreadSchedule-II, state of the table COMPANY is given in Table Ic. It is evident from output of ThreadSchedule-I and ThreadSchedule-II given in Table Ib and Table Ic respectively that final salary of 'Bob' is non-deterministic. Since both threads are accessing same rows of the table, final state of the database depends on the schedule in which two operations are performed. Such operations are dependent operations. In order to test dependent operations, we have to consider all possible schedules involving all active processes at that time. The state space of program with dependent statements is shown in Fig. 5.

In order to differentiate between dependent and independent processes, lets consider same operations addBonus and updateSalary with test inputs (input 2) [maxSalary=10000, Bonus=500] and [increment=2000, loc='California'] respectively. Given the same initial state as in Table Ia, consider

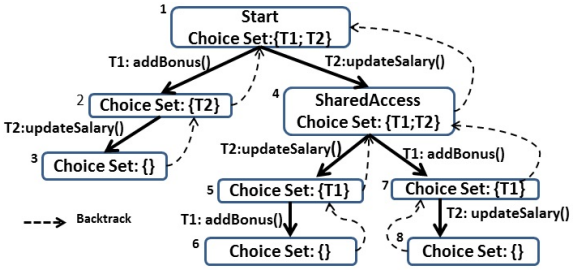


Fig. 5: State Space of the Program with Dependent Statements

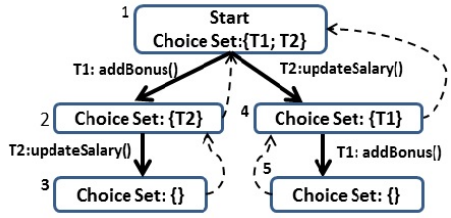


Fig. 6: State Space of Program with Independent Statements

two threads executing ThreadSchedule-I and ThreadSchedule-II (Fig. 3 and Fig. 4).

With ThreadSchedule-I, only one row of the table COMPANY with employee ID 2 is modified when Thread 1 executes addBonus operation. After which, Thread 2 performs updateSalary operation updating the salary of employee with ID 1. After completion of execution of both threads, table COMPANY will be modified to Table Id. We can see that both Thread 1 and Thread 2 operate on different rows of the same table. Now, consider ThreadSchedule-II (Fig. 4) where Thread 2 modifies the salary of employee with ID 1 whereas Thread 1 adds bonus to the salary of employee with ID 2. After successful execution of both operations, it is evident from state of table COMPANY given in Table Id that both threads accessed different rows of the table. The state space of program with ThreadSchedule-I & II is given in Fig. 6.

As it can be seen from the results, no matter in which order these processes are executed, the final state of database remains same for this input. Such processes are independent processes and there is no use to explore both schedules as it would only increase the state-space of the system under test.

The naïve approach to consider all possible schedules of database accesses by different threads will lead to state explosion, especially for a program with larger number of concurrent operations. The state space of program with ThreadSchedule-I in Fig. 3 using naïve approach can be given as in Fig. 7. In our work, we explore different schedules based on dependency relation of operations on each other. Hence, we reduced the state space to be explored by executing only one interleaving of processes, if they are independent. In example given above, state space of program for dependent operations is reduced to Fig. 5, which is further reduced to the Fig. 6 for independent operations, exhibiting significant reduction in state space of program.

TABLE I: STATE OF COMPANY TABLE

(a) INITIAL STATE

ID	Name	Location	Salary
1	Bill	California	16000
2	Bob	Texas	10500
3	Alice	Norway	14000

(b) AFTER EXECUTION OF THREAD SCHEDULE-I WITH INPUT 1

ID	Name	Location	Salary
1	Bill	California	16000
2	Bob	Texas	13000
3	Alice	Norway	14000

(c) AFTER EXECUTION OF THREAD SCHEDULE-II WITH INPUT 1

ID	Name	Location	Salary
1	Bill	California	16000
2	Bob	Texas	12500
3	Alice	Norway	14000

(d) AFTER EXECUTION OF THREAD SCHEDULE I & II WITH INPUT 2

ID	Name	Location	Salary
1	Bill	California	18000
2	Bob	Texas	11000
3	Alice	Norway	14000

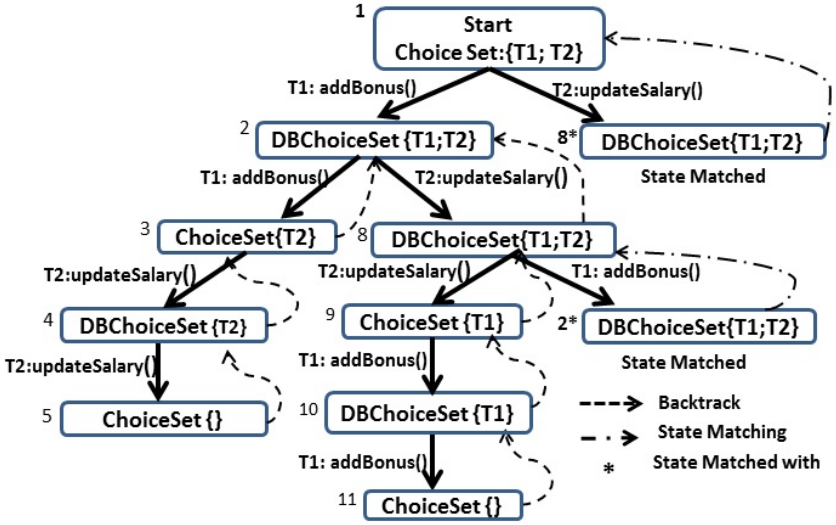


Fig. 7: State Space of Program due to DB Access Choice Points

III. BACKGROUND

A. PostgreSQL

We have used an open source database PostgreSQL. Its relational schema is a finite set of relations used to store data for the applications. Each relation or table has certain attributes. A record is an ordered list of attribute values and representation of a row of the table. In PostgreSQL, every record in a relation has a unique row identifier named `'ctid'` which is a system attribute and its value is assigned by server itself, once the record is inserted for the very first time, or whenever it is updated. `'ctid'` is mapped to $RowId_i$ which uniquely identifies the i^{th} record of the relation r.

Our program communicates with PostgreSQL using PL/pgSQL which is PostgreSQL implementation of the declarative structured query language (SQL). PL/pgSQL provides a set of statements to perform certain operations on the relations of the database. We focus on a simple statements that allow querying, insertion, deletion, or updates in the tables of the database. SELECT, INSERT, DELETE and UPDATE statements are used to perform these operations. In general, SELECT statement is a read only operation, as it returns records in a relation that satisfies the condition c specified in the 'WHERE' clause, whereas INSERT, DELETE and UPDATE statements performs write operation as they modify the state of the database. DELETE and UPDATE statements modify the database state if condition c in 'WHERE' clause is satisfied.

B. Java PathFinder

Java PathFinder is a model checker which explores different interleavings of threads by selecting a thread non-deterministically from a set of live threads. In order to explore all possible schedules, JPF uses the choice generator (to generate non-deterministic choices) on thread operations. JPF generates schedules on the basis of calls to initialization and termination of these threads by executing the statements of the thread until the thread finishes the execution.

JPF executes Java byte code instructions from the application on its own Virtual Machine (VM). JPF has listeners which get notified by the VM whenever a specific operation is performed. These listeners can be used to perform a specific operation and/or to further control the execution of the program. JPF explores states of the program along all possible paths and whenever end of path is reached, it backtracks to explore other unexplored paths.

C. Partial Order Reduction

Partial Order Reduction (POR) is an optimization technique that exploits the fact that many paths are redundant, as they are formed due to the execution of independent transitions in different orders. In the case of database accesses, two operations are independent if both operations produce the same result and database state, regardless of the order in which they are executed.

Transition is basically the sequence of statements executed by a thread without interruption. There are different approaches to define transitions in terms of database accesses that range from coarse to fine grained approaches. Coarse grained approach considers a set of database accesses (which might be multiple) by single thread as a transition, taking thread as a component. It explores all possible interleavings of components.The naïve, fine grained approach on the other hand, considers every database access as a transition. It trivially considers any two database accesses to be dependent and exhaustively explores the entire transition graph which gives rise to the problem of state space explosion.

POR techniques identify dependent transitions and explore only a subset of the paths that are executed by the naïve approach. Thus we need to find all such cases where database access is independent to avoid checking of redundant paths. POR can be applied at different granularity levels from table, record to attribute (cell) level. We applied partial-order reduction at record level of the table using $RowId_i$. However, attribute level can be easily done by comparing columns of the tables as well.

IV. Technique

We have modeled concurrent operations with a multi-threaded application. In order to identify when a thread is about to execute a statement that involves database access, we simply identify a database access as an invocation of Java Statement class methods. Whenever we find such an invocation we perform a dependency analysis. In case of dependent database accesses we mark the accesses as shared. Doing this for threads makes sure that if two threads have shared database accesses then we will explore the other interleaving as well to test non-deterministic behavior of the application. Our partial order reduction process has four parts:

1) Identify database access during state exploration.
2) Get instructions that are scheduled for execution and query database.
3) Perform dependency analysis and mark dependent database accesses.
4) Add JPF choice point (to explore alternative choices) for dependent database accesses.

The flow of our partial order reduction process is given in Fig. 8.

A. EPOR Handling of DB Operations

We run SQL statements in PostgreSQL to extract $RowId_i$ and then find the dependencies by analyzing the output of these queries. For each SQL statement Q executed on the database,

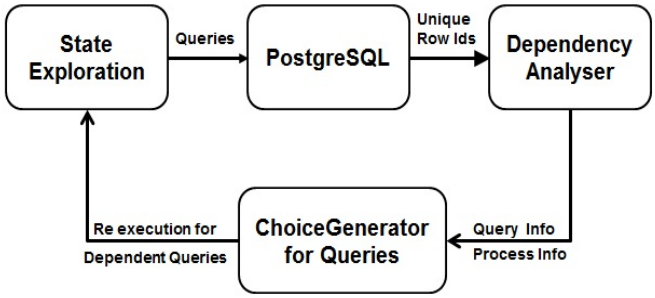


Fig. 8: Partial Order Reduction Process

where $Q \in \{$ S = Select , D= Delete, I = Insert, U = Update $\}$, we define the semantics of the output of these database accesses as follows:

1) The **SELECT** statement, returns a set of unique row identifiers of the rows that satisfy the condition c in 'WHERE' clause or all rows present in the relation in absence of the condition c. This can be given as $rowId(S) = \{RowId_i | i \in \{1, 2, ..., n\}\}$.
2) The **INSERT** statement returns a unique row identifier of newly inserted rows if insertion is successful, that is, $rowId(I) = \{RowId_i\}$.
3) The **UPDATE** statement returns the updated unique row identifier of the record, if it satisfies the condition c in the 'WHERE' clause, represented as $rowId(U) = \{RowId_i | i \in \{1, ..., n\}\}$.
4) The **DELETE** statement returns the unique row identifier of the deleted record if it satisfies the condition c in the 'WHERE' clause, represented as $rowId(D) = \{RowId_i | i \in \{1, ..., n\}\}$.

B. Instrumentation of PostgreSQL

To check the dependency conditions, we have extracted unique row identifiers of the records of the relation through instrumentation of PostgreSQL. After getting two database accesses for same relation by different threads, we actually execute queries by connecting to PostgreSQL server. Whenever PostgreSQL encounters an SQL statement, it prepares many possible execution plans to process the statement. It selects the optimal plan after estimating the cost of each plan and then executes it. During execution of the plan, in the initialization phase of the list of query nodes, we append an extra node to the list to get $RowId_i$ of the records returned by the query. After complete execution of the query, we gather $RowId_i$(s) of the accessed records returned by the query.

C. Detection of Operational Dependencies

For database applications, we track the dependencies among SQL operations at record level. We have defined conditional dependencies among SQL operations as in Table II, in form of pair of database operations. Database accesses are represented by the first letter of the SQL operation. For example, (S_1, U_2) defines sequence of database operation, i.e., SELECT by first process followed by UPDATE by second process. Dependence

TABLE II: DEPENDENCY RELATIONS FOR DATABASE ACCESSES

DB Operation	Dependence Relation	Execution Order	Conditions Used by EPOR to compute dependence	Dependence Relation (DPF)[a] [8]
(S_1, S_2)	No	-	-	-
(I_1, I_2)	No	-	-	-
(D_1, D_2)	No	-	-	-
(S_1, U_2)	Yes	(S_1, U_2, S_{1a})	rowId(S_1) $\cap$ rowId(S_{1a}) $\neq$rowId(S_1)	Conditional Dependence
(U_1, S_2)	Yes	(S_2, U_1, S_{2a})	rowId(S_2) $\cap$ rowId(S_{2a}) $\neq$ rowIdS_2	Conditional Dependence
(S_1, I_2)	Yes	(S_1, I_2, S_{1a})	rowId(S_1) $\cap$ rowId(S_{1a}) $\neq$ rowIdS_1	Always Consider Dependent
(I_1, S_2)	Yes	(S_2, I_1, S_{2a})	rowId(S_2) $\cap$ rowId(S_{2a}) $\neq$ rowIdS_2	Always Consider Dependent
(S_1, D_2)	Yes	(S_1, D_2, S_{1a})	rowId(S_1) $\cap$ rowId(S_{1a}) $\neq$ rowIdS_1	Conditional Dependence
(D_1, S_2)	Yes	(S_2, D_1, S_{2a})	rowId(S_2) $\cap$ rowId(S_{2a}) $\neq$ rowIdS_2	Conditional Dependence
(U_1, U_2)	Yes	(U_1, U_2)	rowId(U_1) $\cap$ rowId(U_2) $\neq \emptyset$	Conditional Dependence
(I_1, U_2)	Yes	(I_1, U_2)	rowId(I_1) $\cap$ rowId(U_2) $\neq \emptyset$	Conditional Dependence
(D_1, U_2)	Yes	(D_1, U_2)	rowId(D_1) $\cap$ rowId(U_2) $\neq \emptyset$	Conditional Dependence
(D_1, I_2)	Yes	(D_1, I_2)	rowId(D_1) $\cap$ rowId(I_2) $\neq \emptyset$	Conditional Dependence

[a]Conditional Dependence are less precise except at attribute granularity (See Section V)

Relation 'Yes' or 'No' represents that both accesses are dependent or independent of each other, if they satisfy the condition given in 'Conditions used by EPOR' column, with the given execution order. The last column of Table II lists dependency relationship defined by the DPF [8]. Conditional dependence means that if the condition is satisfied by the function defined in DPF, only then, the pair of operations is dependent otherwise independent. Symbol '-' shows that there is no need to specify condition as both operations are independent of each other.

There are certain sequences of database operations that are always independent, e.g., two deletes represented as (D_1, D_2) are always independent, as the combined effect of both deletes remain same, irrespective of their order of execution. Similarly, two inserts (I_1, I_2) in a database does not result in non-deterministic database state. Since the output of the insertion operation is an addition of a new record in the table, so after execution of (I_1, I_2) or (I_2, I_1) database will have maximum two new records. It is quite obvious for two statements querying database that both will be independent statements as they are not affecting the resulting database state.

In case of two updates (U_1, U_2), database access could be dependent on the order of the execution. For example, there could exist one record R_o which satisfies the condition c_1 of U_1 and get modified such that updated value of R_o is R_u. When U_2 is executed, the modified value R_u does not satisfy the condition c_2 of U_2, whereas the original value of record R_o satisfies the condition c_2 of U_2. In this situation, the order of execution of U_1 and U_2 can lead to different database states. Thus, this operation is not independent and in order to test it against given specifications, we need to explore both schedules, that is, U_1 followed by U_2 ($U_1 \rightarrow U_2$) and U_2 followed by U_1 ($U_2 \rightarrow U_1$). In order to identify the dependent operations, we check the conditions specified in Table II. For example, if two updates are dependent then they must be updating the same records. So intersection of unique row identifiers returned by each of these two queries will not be an empty set.

In operations that involve one SELECT statement, e.g., (U_1, S_2), execution order (S_2, U_1, S_{2a}) shows firstly we run SELECT operation followed by UPDATE, then we rerun SELECT given as S_{2a} to apply dependency condition.

D. Implementation Details

As we mentioned earlier, we model parallel processes as different threads. Our partial order reduction technique builds on top of JPF. We trapped calls to methods of Statement class by intercepting all method invocations that access the database. JPF notifies our listener when it encounters executeQuery, executeUpdate or execute method of the Statement class. We extract the statement being executed by the call and parse it to obtain the list of tables accessed and access type of statement (READ or WRITE). We have defined a list structure to store the information we extracted. Every time we encounter these methods we traverse through our structure to find a pair of threads t_i and t_j, such that the instruction to be executed by each thread is execute method call of a Statement object. It then compares thread information along with their access type to perform dependency analysis. If both operation/queries performed by threads are dependent i.e. state of the database can be non-deterministic, then we mark the set of statements as shared in our data structure as described in Fig. 9.

In order to call methods of an object, Java uses the bytecode instruction 'INVOKEVIRTUAL'. We intercept the execution of 'INVOKEVIRTUAL' instruction every time it is called. For Statement methods, we check, for the running thread, if there exists any database access shared with any other thread. In presence of shared database access with one or more threads, we add a choice point for the model checker at this point, with all threads sharing database access as possible choices. This makes sure that the model checker makes schedules by pre-empting threads on these method calls. Pseudocode is shown in Fig. 10.

```
1  define C_dep as conditional dependence;
2  L ← dbAccessInfoList;
3  T_info ← Get information of current thread t;
4  M ← Invoked method from T_info;
5  if (M is a method of Statement class) then
6      Q ← GetDBStatement(T_info);
7      Rel ← TablesAccessed(Q , T_info);
8      Access ← AccessType(Q);
9      for each element E in L do
10         if (E ≠ T_info) AND (E.Rel=T_info.Rel) AND
           ((E.Access OR T_info.Access) ≠ READ) then
11             (R_i, R_j) ← ExecuteQuery(E.Q, T_info.Q);
12             C_dep ← DependencyAnalysis(R_i, R_j);
13             if (C_dep) then
14                 MarkDBAccessShared(E, T_info);
15             end
16         end
17     end
18     if ( ! (isPresent(T_info, L_d))) then
19         add T_info to L;
20     end
21 end
```

Fig. 9: Marking Dependent Transition Algorithm

```
1  define T_curr as active thread;
2  B ← ProgramByteCode;
3  L ← dbAccessInfoList;
4  L_d ← dependentAccessInfoList;
5  I ← read Instruction from B;
6  while (B ≠ end of Bytecode) do
7      if (I = 'INVOKEVIRTUAL') then
8          T_info ← ThreadInfo of T_curr;
9          if (L ≠ 'NULL') then
10             if (isPresent(T_info , L_d)) then
11                 if (!(ChoicePoint)) then
12                     add ChoicePoint;
13                 end
14             else
15                 if (isDependent()) then
16                     add T_curr to L_d;
17                 end
18             end
19         else
20             add T_curr to L;
21         end
22     end
23     I ← read next Instruction from B;
24 end
```

Fig. 10: Partial Order Reduction Algorithm

E. Backtracking and Database State Restoration

Execution of queries in a sequence will change the state of the database. For example, UPDATE followed by SELECT ($U \rightarrow S$) will change the database state. If SELECT and UPDATE are dependent accesses then we will explore other interleaving as well. Before exploring $S \rightarrow U$, it is important to restore the database to its original state i.e. state that was before execution of $U \rightarrow S$. For backtracking the database state, we exploit the savepoint and rollback mechanism of the database, so we can roll back the database state at a backtrack point. For each state in JPF, we set a savepoint. We have mapped JPF state identifiers to savepoint in order to insert savepoint in database. When JPF backtracks restoring the memory state, we rollback to the respective savepoint, by using mapping of state identifiers.

F. Challenges and their solution

There are significant challenges that we face during implementation and testing of database applications. We customize JPF to address all mentioned challenges and enable the exploration of database applications.

- One of the challenges that we faced during implementation phase was that JPF cannot simulate classes that use native code for their functionality, unless a model is written for the classes to translate objects sent and received to native code. Database accesses need drivers and in order to access and use these drivers, Java uses classes, that are not modeled in JPF. In order to get around this problem, we wrote wrappers classes required to communicate with database. The wrappers provide the same interface as the Java SQL classes but do not connect to the database like the actual classes. Instead these classes simulate database access.
- JPF sees accesses to shared memory objects as the only source of non-determinism. Since database does not load in its memory during program execution, JPF is unable to identify non-determinism in the processes that are accessing database. We have addressed this challenge by defining our own data structure in JPF to store information about shared database accesses.
- JPF does not see shared access to database. So, it does not consider transitions due to shared database access, as the scheduling points. Shared database accesses are the key scheduling points for database applications. Thus, in order to explore their possible schedules, we add a choice generator at these points.

TABLE III: NUMBER OF STATES GENERATED

No. of DB Operations	No. of States		No. of States	
	Dep	Indep	Dep(DPF)	Indep(DPF)
2	12	10	15	9
3	14	12	21	14
4	33	22	27	18
5	59	44	70	38
6	56	40	183	40
7	111	53	257	48

V. Evaluation

A. Configurations and Benchmarks

Each thread of our implementation has its own local variables and can communicate with database through SQL statements. We experimentally evaluate the effect of Partial Order Reduction on the state space of program. We have shown that the number of states explored decreases from naïve exhaustive exploration, with much reduction. We have also compared our results with Database PathFinder (DPF)'s POR technique and shown that for record level granularity, our technique is more precise, as in some cases it gives better reduction in number of states and instructions executed.

All experiments were performed on a machine with a 3-core Intel Core i3-370M processor and 4GB of main memory, running Ubuntu Linux 14.4 and PostgreSQL 9.3.

Our set of benchmarks includes one live Java application PetClinic4, which is an official sample distributed with the Spring Framework[3] and implements an information system to be used by a veterinary clinic to manage information about veterinarians, pet owners, and pets. Secondly, our own created example of Employee Management System. We have implemented several operations to check effectiveness of our technique. Two operations are presented above in Section II.

PetClinic is basically a web based application. We modeled it by designing an interface through which we send commands to Java code containing application logic. Since PetClinic does not include concurrent cases (test case for two or more threads), we created concurrent test cases by combining operations in such a way that each operation is executed by one thread as one isolated process. Since, Java PathFinder works on a particular input, so we had to create set of inputs for our test applications. In order to have unbiased results we considered a range of values as input for test cases.

B. Discussion

We executed a number of threads with varied number of database accesses per thread for a range of input values. We encountered dependent and independent database accesses during execution. For independent operations number of states explored were less than that of dependent operations. It can be seen from results given in Table III that with increase in number of database accesses, the state space of program also increases for dependent operations.

In order to evaluate performance of our technique, we compared EPOR with DPF for dependent and independent database accesses. We have seen that DPF approach takes pessimistic decisions about database accesses especially when it encounters a pair of INSERT and SELECT statement by considering it to be dependent access. However, we can also note that most of the times the number of states generated by our algorithm for independent operations were more than those generated by DPF. It is due to the fact that our algorithm does not consider disjoint attributes of same row as independent. So there is a possibility of marking accesses as dependent even

[3]http://springsource.org/

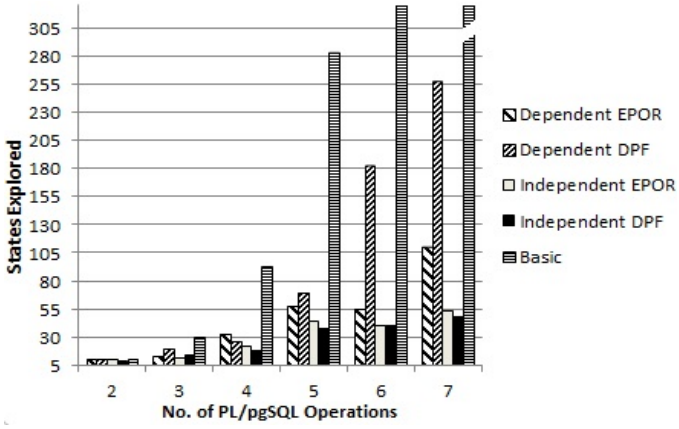


Fig. 11: State space of Program for Dependent and Independent DB Accesses

when two disjoint attributes of same row are accessed, eventually generating more states. It is evident from Fig. 11 that increase in number of states for basic approach is exponential. There is significant decrease in states explored by EPOR for dependent operations which shows that our algorithm returns more precise results at record level granularity. Although it is quite trivial to implement same logic at attribute level, since we cannot improve on attribute level as compared to DPF, we restricted our scope to the record level granularity.

Summary of results for PetClinic given in Table IV shows that our algorithm requires less time for the verification of independent accesses than that required by dependent database accesses. Hence, for programs with larger number of independent accesses as compared to dependent accesses will give significant reduction in state space of program.

Result of our second example Employee Management System depicts significant reduction in number of state of system under test. In general, our technique gives on average, 1.4x reduction in state space of program. If we compare effectiveness of our algorithm for dependent and independent database access, we can see that if our program contains independent accesses then verification takes less time giving average speed up of 2.8x. With the increase in number of independent database accesses as well as number of concurrent operations, our technique will improve on overall speedup and reduction in state space of program as shown in Table V.

While comparing with DPF, we present result of experiments that involves states explored, the number of instruction executed and the execution time. Results are given in Table VI. The number of instructions executed by DPF is more than our algorithm due to the fact that many independent accesses are marked dependent by the DPF algorithm, in Fig. 12c. However, our algorithm takes more time as given in Fig. 12b, for execution as number of processes increase because EPOR works on record level only and also due to re-execution which would only pay off for larger programs. For programs with less than 75 per cent database operations as dependent and remaining as independent will perform better in terms of

TABLE IV: SUMMARY OF RESULTS FOR PETCLINIC FOR DEPENDENT AND INDEPENDENT OPERATIONS

No. of Queries	Dep. Queries	Indep. Queries	States EPOR	States DPF	Time EPOR(ms)	Time DPF(ms)	Memory (MB)	Instructions EPOR
2	0	2	10	10	1200	1000	69	5277
2	2	0	12	12	1200	1000	69	5506
3	2	1	12	14	1400	1000	69	5634
3	0	3	10	14	1600	1000	69	5405
4	0	4	22	24	4000	4000	69	9136
4	2	2	26	29	6000	5000	99	9983
4	3	1	33	32	8000	7000	99	10792
5	2	3	50	68	19000	18000	99	18793
5	5	0	53	70	20000	19000	99	18826
7	2	5	100	182	72000	48000	129	40564
7	5	2	111	196	74000	52000	299	44888

TABLE V: SUMMARY OF RESULTS FOR EMPLOYEE MANAGEMENT SYSTEM

No. of DB Accesses	Independent Operations Time (ms)	Dependent Operations Time(ms)	States Indep.	States Dep.	Speedup	Reduction
3	2000	3000	18	23	1.5	1.27
4	23000	85000	42	66	3.7	1.57
5	50000	168000	52	86	3.36	1.65
6	70000	189000	69	93	2.7	1.34

execution time and reduction in unnecessary states of program due to the fact that for independent accesses only one schedule out of many will be explored.

Since our code for JPF is generic, our technique can easily be adapted to other database systems with instrumentation. Instrumentation of RDBMS is trivial if source code is available. Moreover, in presence of transactions, we can track transaction's commit in execute method and then apply EPOR on all queries present in the transaction by considering it a single operation.

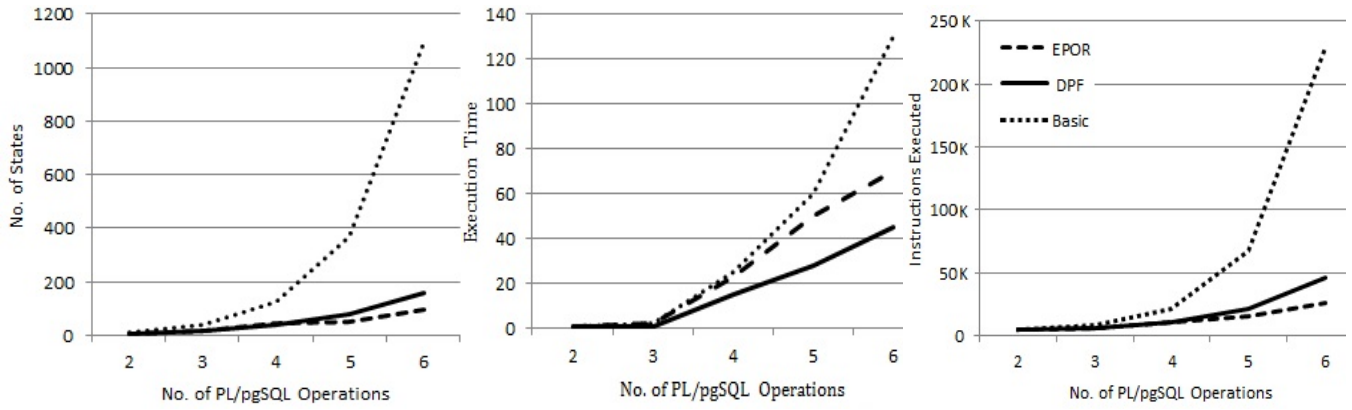
VI. RELATED WORK

In closely related work, Gligoric and Majumdar [8] designed and implemented DPF (Database PathFinder), an explicit-state model checker for database-backed web application also built on top of the JPF model checker. They presented several implementation choices such as stateful vs stateless search, state storage for state restoration, backtrack strategies and dynamic partial order reduction. They presented multiple strategies for partial order reduction at different granularity levels by analyzing SQL statements and their dependencies on each other. They showed the fine grained partial order reduction reduces the number of states most. The two most fine grained approaches considers dependencies based on rows and attributes. Our approach differs in two ways. First, we find the set of affected rows from the query engine in the database. Determining affected rows from inside the database is more precise than the pessimistic approach of determining what might be affected from the SQL statement. This also makes the technique more robust. Second, for operations that do not form disjoint sets, they pessimistically declare them dependent, however we proceed to execute and check the alternate order in the database to see if there really is an affect given the data stored in the database. For example they considered select and insert operation always dependent as insertions and selection operations does not form disjoint sets. Our approach is more effective in a way that we find dependencies over database state by re-executing queries for dependency analysis resulting in less number of false positive and more precise partial order reduction.

DPF also introduces a further mode where operations which do not overlap in attributes are considered independent. This is an orthogonal improvement that can be easily combined with our approach. In cases where one of the operations is a SELECT, we are still able to identify that the operations are independent because of our re-execution albeit giving the cost of re-execution which was not necessary if attribute detection is added. Lastly, DPF does not handle complex queries like joins or sub-queries. We have not implemented complex queries, however, our technique enables much easier implementation of joins because we do not need to process SQL statements and deduce affected rows from them. Rather, we can use our existing instrumentation in the database to find rows affected in both tables and find disjoint sets. We aim to achieve that in future work.

Paleari et al. [10] found dependent operations through a log from a single execution, ignoring program semantics, and did not perform model checking of other possible orders. This made it label many operations as dependent when they really were not. Zheng and Zhang [11] used static analysis to find race conditions when accessing external resources. Static analysis inherently has more false positives (labeling more operations dependent). Compared to both these techniques, our technique has less false positives since we are based on model checking but requires actual execution.

Emmi and Majumdar [12] presented concolic execution to generate both input data for the program as well as suitable database records to systematically explore all paths of the program, including those paths whose execution depend on data returned by database queries. Marcozzi et al. [13] describe symbolic execution of SQL statements along with other constraints in program for generating test inputs in order to



(a) Number of States Explored (b) Execution Time for State Space Exploration (c) Number of Instructions Executed

Fig. 12: Comparison of EPOR, DPF and naïve approach

TABLE VI: MODEL CHECKING EMPLOYEE MANAGEMENT SYSTEM USING DPF, EPOR AND BASIC APPROACH

No. of Queries	Ins. EPOR	Ins. DPF	Ins. Basic	States EPOR	States DPF	States Basic	Time (ms) EPOR	Time (ms) DPF	Time (ms) Basic
2	4154	4154	5050	4	6	12	1000	1000	1000
3	5671	5622	8378	18	16	40	2000	1000	2000
4	9874	9867	20996	42	38	122	23000	15000	25000
5	15209	20488	66940	52	78	366	50000	28000	60000
6	25681	46001	229550	93	158	1096	70000	45000	130000

test database application. Other researchers have also used symbolic techniques to test databases [14]–[18]. Symbolic analysis is orthogonal to our technique and can be combined with it.

Software Model checking is an effective technique for verification of concurrent applications [2], [3], [9]. The basic technique of model checking is however not directly applicable to database applications. Partial order reduction in model checking is concerned with removing transitions that work on different shared variables. Flanagan and Godefroid presented partial order reduction that depends on dynamically detected dependencies [5] and showed that it performs much better than static detection. We apply and optimize dynamic partial order reduction for database operations.

Model checking has also been used for web applications [19]–[21] to model check interleaving of web requests but not of individual database operations. Thus, these techniques cannot detect issues when two requests are running concurrently and their individual database operations are not performed in a transaction. Web applications that avoid one transaction per request model to achieve scalability have a possibility of races between database operations. Our technique will correctly model check such applications.

Database testing also involves checking of integrity constraints on schema. G.M. Kapfhammer [22] presented a technique for testing of relational schema. It can handle multiple databases as it works on SQL based constraints that are related to database schema and not stored in real memory.

VII. CONCLUSIONS

In this paper, we have developed and tested a novel technique for more effective partial order reduction for model checking database applications. Our technique is based on model checking and builds on top of JPF which is a well-known model checker for Java applications and on PostgreSQL database engine. Two fundamental limitations in model checking are the exponential number of schedules and input dependence. Our proposed optimizations tries to reduce the number of explored schedules by finding dependent operations using backtracking database state, finding affected rows, and re-executing when necessary. We improve upon prior work (Database PathFinder) by performing a more precise analysis that is able to identify more independent operations and thus reduce the number of states further. In our evaluation, we show the reduction in number of states explored and instruction executed to explore state space of program compared to Database PathFinder.

In future work, we intend to remove dependence on program input by introducing a symbolic database. By using a symbolic database, we would be able to identify possible input dependencies and explore each possibility separately. Furthermore, we intend to support complex SQL queries like table joins in future work. Extending Database PathFinder to handle complex queries is difficult and would require taking many pessimistic decisions because it analyzes SQL statements directly. However, extending our technique to handle complex query is easier because we depend on the database query processing engine to find the rows that are being accessed for various attributes directly.

REFERENCES

[1] P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency control and recovery in database systems*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1987.

[2] C. Baier and J.-P. Katoen, *Principles of model checking*. The MIT Press, 2008.

[3] G. J. Holzmann, "The model checker SPIN," *IEEE Transactions on software engineering*, vol. 23, no. 5, pp. 279–295, 1997.

[4] P. Godefroid, "Model checking for programming languages using verisoft," in *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 1997, pp. 174–186.

[5] C. Flanagan and P. Godefroid, "Dynamic partial-order reduction for model checking software," in *ACM Sigplan Notices*, vol. 40, no. 1, 2005, pp. 110–121.

[6] P. Godefroid, J. van Leeuwen, J. Hartmanis, G. Goos, and P. Wolper, *Partial-order methods for the verification of concurrent systems: an approach to the state-explosion problem*, 1996, vol. 1032.

[7] E. M. Clarke, O. Grumberg, M. Minea, and D. Peled, "State space reduction using partial order techniques," *International Journal on Software Tools for Technology Transfer*, vol. 2, no. 3, pp. 279–287, 1999.

[8] M. Gligoric and R. Majumdar, "Model checking database applications," in *Tools and Algorithms for the Construction and Analysis of Systems*, 2013, pp. 549–564.

[9] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda, "Model checking programs," *Automated Software Engineering*, vol. 10, no. 2, pp. 203–232, 2003.

[10] R. Paleari, D. Marrone, D. Bruschi, and M. Monga, "On race vulnerabilities in web applications," in *Proceedings of the 5th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, ser. DIMVA '08, 2008, pp. 126–142.

[11] Y. Zheng and X. Zhang, "Static detection of resource contention problems in server-side scripts," in *Proceedings of the 34th International Conference on Software Engineering*, ser. ICSE '12, 2012, pp. 584–594.

[12] M. Emmi, R. Majumdar, and K. Sen, "Dynamic test input generation for database applications," in *Proceedings of the 2007 international symposium on Software testing and analysis*, 2007, pp. 151–162.

[13] M. Marcozzi, W. Vanhoof, and J.-L. Hainaut, "Testing database programs using relational symbolic execution," Technical report, University of Namur, Tech. Rep., 2014.

[14] K. Pan, X. Wu, and T. Xie, "Database state generation via dynamic symbolic execution for coverage criteria," in *Proceedings of the Fourth International Workshop on Testing Database Systems*, 2011, p. 4.

[15] S. A. Khalek and S. Khurshid, "Systematic testing of database engines using a relational constraint solver," in *Software Testing, Verification and Validation (ICST), 2011 IEEE Fourth International Conference on*, 2011, pp. 50–59.

[16] K. Pan, X. Wu, and T. Xie, "Generating program inputs for database application testing," in *Automated Software Engineering (ASE), 2011 26th IEEE/ACM International Conference on*, 2011, pp. 73–82.

[17] G. Wassermann, D. Yu, A. Chander, D. Dhurjati, H. Inamura, and Z. Su, "Dynamic test input generation for web applications," in *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, ser. ISSTA '08, 2008, pp. 249–260.

[18] V. Felmetsger, L. Cavedon, C. Kruegel, and G. Vigna, "Toward automated detection of logic vulnerabilities in web applications," in *Proceedings of the 19th USENIX Conference on Security*, ser. USENIX Security'10, 2010, pp. 10–10.

[19] B. Petrov, M. Vechev, M. Sridharan, and J. Dolby, "Race detection for web applications," in *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '12, 2012, pp. 251–262.

[20] M. Martin and M. S. Lam, "Automatic generation of xss and sql injection attacks with goal-directed model checking," in *Proceedings of the 17th Conference on Security Symposium*, ser. SS'08, 2008, pp. 31–43.

[21] S. Artzi, A. Kiezun, J. Dolby, F. Tip, D. Dig, A. Paradkar, and M. D. Ernst, "Finding bugs in web applications using dynamic test generation and explicit-state model checking," *IEEE Trans. Softw. Eng.*, vol. 36, no. 4, pp. 474–494, Jul. 2010.

[22] G. M. Kapfhammer, P. McMinn, and C. J. Wright, "Search-based testing of relational schema integrity constraints across multiple database management systems," in *Software Testing, Verification and Validation (ICST), 2013 IEEE Sixth International Conference on*. IEEE, 2013, pp. 31–40.