

An Integrated Database Programming and Querying Language with Support for Simultaneous Processing

Markus Kirchberg

Information Science Research Centre and Department of Information Systems,
Massey University, Private Bag 11 222, Palmerston North 5301, New Zealand

Email: M.Kirchberg@massey.ac.nz

Abstract

The integration of query languages and programming languages is still a central problem of database research. In particular, issues with respect to the impedance mismatch are of interest. In this paper, we introduce features of a newly developed integrated database programming and querying language (i)DBPQL, which unites properties from object-oriented programming languages, query languages and database programming languages. Our main focus is on those integrated language properties that support simultaneous processing and their implications on other database system components.

1. Introduction

The relationship between query languages (QLs) and programming languages (PLs) has been studied for decades. While embedded approaches suffer from the impedance mismatch, integrated approaches do not. The majority of proposed integrated approaches either represent a PL with added QL constructs (e.g. DBPL [2]) or a QL with added programming abstractions (e.g. Oracle PL/SQL [4]). Only a few research projects (e.g. LOQIS [8]) have attempted a seamless integration of programming and querying languages. We follow the latter line of research as part of a bigger project [7], which aims to develop a distributed object database system (ODBS) that is based on a sound theoretical framework.

In this paper, our main focus will be on those integrated language properties that relate to simultaneous processing. Edsger Dijkstra defined a general notion of simultaneous processing as follows: ‘*Concurrency occurs when two or more execution flows are able to run simultaneously*’. Thus, it encompasses computations that execute overlapped in time, and which may permit the sharing of common resources between those overlapped computations. This re-

sults in a number of potential advantages, which include:

- Reduction in run-time, increase in throughput and decrease in response time;
- Increase in reliability through redundancy;
- Potential to reduce duplication in code; and
- Potential to solve more real-world problems than with sequential computations alone.

As promising as these advantages may sound, they do not come without potential drawbacks. The use of shared resources, added synchronisation and communication requirements lead to a number of disadvantages, which include:

- Run-time is not always reduced, i.e. careful planning is required;
- Concurrent computations can be more complex than sequential computations;
- Shared data can be corrupted more easily; and
- Communication between tasks is needed.

Thus, while support for simultaneous processing brings many potential advantages, corresponding drawbacks must not be omitted from considerations.

1.1. Contribution

In this paper, we introduce features of a newly developed integrated database programming and querying language (i)DBPQL, which unites properties from object-oriented programming languages, query languages and database programming languages. Our main focus will be on those integrated language properties that support simultaneous processing and their implications on other database system components, in particular, transaction management.

1.2. Outline

The remainder of this paper is organised as follows: Section 2 briefly discusses related work. Subsequently, Section 3 provides an overview of the proposed iDBPQL language. The support of simultaneous processing and corresponding implications are discussed in Section 4 in greater detail. Finally, Section 5 concludes the paper.

2. Related Work

While there are numerous research contributions that address the development of integrated languages and object-oriented (database) environments, we will restrict ourselves to the most relevant publications.

The *Franco-Armenian Data Model (FAD)* [3] has been designed for highly parallel database systems. Besides query operations, the optimisation of programming language constructs, in terms of utilising parallelism, is considered to enhance system performance. FAD provides a rich set of built-in structural data types and operations. It operates on sets and tuples, which can be nested within each other to an unlimited degree. FAD operators mainly utilise pipelining and set-oriented parallelism. For instance, a `filter` operator is proposed targeting sets. In addition, a `pump` operator, which supports parallelism by a divide and conquer strategy, targets aggregate functions. Both operators have influenced how iDBPQL supports simultaneous processing.

FAD was later extended [5] with communication primitives, in particular with asynchronous message passing mechanisms. The resulting language is called *PFAD*, which is restricted to a shared-nothing, distributed model of execution.

3. An Overview of DBPQL and iDBPQL

In a truly distributed ODBS, users are not aware of the distributed nature of the system. In fact, data independence and network and fragmentation transparency are key system properties. Thus, user requests are free of location- and communication-specific information. Only during the compilation, fragmentation, code rewriting, linking and optimisation processes such information is added.

Assume that user requests arrive in the form of DBPQL programs or modules. DBPQL, which is based on the internal language iDBPQL, is a high-level database programming language. DBPQL is a modular language that supports transactions, query processing, generic requests, type, class and object creation / manipulation commands etc. The database system (DBS) component that processes user requests consists of a DBPQL compiler, code optimis-

ers, code rewriters (e.g. to map operations on global objects to local objects) etc. Here, we follow a black box approach assuming that incoming DBPQL modules are transformed into optimised evaluation plans, which are formulated in iDBPQL code [6]. Thus, from an internal point of view, each user request corresponds to an optimised evaluation plan with a number of associated metadata structures. An evaluation plan consists of iDBPQL control flow statements, assignments, (query) expressions, method calls (i.e. references to other evaluation plans), and sub-blocks. High-level concepts such as modules, interfaces, type and class definitions, classes (as code structuring primitives) etc. have been removed. Metadata structures correspond either to the DBS metadata catalogue, which is a collection of compiled database schemata, or to the run-time metadata catalogue, which is a collection of type and class definitions introduced in the source code of the user program. While DBS metadata entries describe persistent, shared data, run-time metadata entries relate to transient, non-shared data. Metadata entries may have further evaluation plans associated. Such plans are attached to all non-abstract behaviour specifications and outline their respective implementations.

Considering iDBPQL metadata entries, we distinguish between values and objects. Values are mutable data items identified by their value while objects are data items that have an immutable object identifier (of an internal type `_OID`) independent of associated values. Types structure values. Supported types include primitive types (i.e. `BOOL`, `CHAR`, `INT`, `NAT`, and `REAL`), the record type, parameterised user types, collection types (including `BAG`, `SET` and `LIST`) and the `NULLable` type. Types are later extended to include reference-types and the `UNION`-type supporting the unification of identical or similar objects. Sub-typing is structural (order, types and names are considered). While behaviour is not inherited, a sub-type can utilise its super-type's behaviour through type mapping. Classes are used to group more complex structures. The structure of a class of objects is defined over existing types, unnamed types (i.e. types without a behaviour that are defined in the class structure itself) and existing classes (either as inheritance or as reference). Classes are templates for creating objects, expose structural properties, allow for the definition of (reverse) references, may have associated behaviour (i.e. instance methods, class methods and object constructors – all of which are represented as an evaluation plan), support multiple inheritance, and may have associated, system-maintained collections through which access to all objects of a class and its sub-classes is possible.

Evaluation plans consist of control flow statements, assignments, expressions, method calls, and sub-evaluation blocks. Common programming abstractions (e.g. object creation statements, assignments, conditional statements, various loops and sub-routines), and query language con-

structs (e.g. selection, projection, navigation, join, and order-by) are supported. The integration of both concepts mainly evolves around collections. Evaluation blocks are used to group statements together, form atomic execution units, model local and distributed transactions, support independent or multi-threaded processing etc. Simultaneous processing is utilised to enhance performance. While the transaction management system (TMS) allows different transactions to execute simultaneously (i.e. inter-transaction concurrency), iDBPQL also supports two expressions that explicitly request simultaneous execution. The latter may occur at the transaction-level (i.e. inter-transaction concurrency) or at the operation-level (i.e. intra-transaction concurrency).

In addition, there exists an *iDBPQL library*, which contains definitions and implementations for all built-in iDBPQL features.

4. iDBPQL and Simultaneous Processing

Concurrency and, to some degree, also parallelism are supported by the iDBPQL language. While there are various types of simultaneous processing, e.g. multi-programming, multi-processing, control parallelism, process parallelism, data parallelism, multi-threading, distributed computing etc. (some of which refer to the same processing type), only some of them are supported explicitly, but others are utilised implicitly.

iDBPQL mainly utilises time sharing (i.e. concurrency in terms of multi-tasking and multi-threading) as well as MIMD and SIMD multi-processing (i.e. true parallelism). While the transaction management system allows different transactions to execute simultaneously (i.e. inter-transaction concurrency or parallelism), iDBPQL also supports two expressions that explicitly request simultaneous execution. The former is discussed in Section 4.1. The latter may happen on the transaction-level (i.e. inter-transaction concurrency) or the operation-level (i.e. intra-transaction concurrency). Section 4.2 discusses the support of simultaneous processing in greater detail. Finally, individual operations may be implemented in ways that simultaneous processing is utilised (i.e. intra-operation concurrency). However, this form of processing is not discussed in this paper since it is more closely associated with the implementation of iDBPQL.

4.1. Implicit Inter-Transaction Concurrency

As typical for DBSs, independent transactions are executed simultaneously whenever possible. The two most commonly considered properties, affecting the degree of interleaving, are serialisability (i.e. conflict-serialisability)

and recoverability. This, of course, is also the case for transactions in iDBPQL that stem from different user programs, i.e. originate from different main evaluation plans. The TMS scheduler determines the degree of interleaving of operations of such transactions. Transactions that belong to the same main evaluation plan may be executed serially or interleaved. The programmer has a greater influence over the mode of execution. Corresponding details are outlined next.

4.2. Support for Explicit Concurrency

Inter- and intra-transaction concurrency may be specified explicitly. iDBPQL provides two different control flow expressions that imply concurrency.

On one hand, it can be specified that a block of statements may be executed independently from its surrounding statements (i.e. time sharing or MIMD). Thus, different, independent statements are processed at the same time. When specified within a loop statement, this time sharing or MIMD approach is mixed with SIMD. The general syntax for such a specification is as follows:

```
"INDEPENDENT", "DO", Statements, "ENDDO";
```

Whether or not independent execution results in multi-tasking, single-threaded or multi-threaded execution is indicated by the compiler and / or decided at execution time.

Let us consider some examples. First, we consider the simultaneous execution of two independent statements, which belong to the same transaction (i.e. utilising intra-transaction concurrency).

```
01 PUBLIC EVALPLAN ViewProfile ( ) {
02     doSomething;
03
04     LABEL i1 : INDEPENDENT DO
05         doSomethingIndependently;
06     ENDDO;
07
08     doSomethingElse;
09
10     WAIT i1;      // synchronisation of the
11                  // main execution stream with
12                  // the independent stream
13
14     doMore;
15 }
```

From line 01 to line 04 execution has been serial. In line 04, an independent execution block is declared. Together with this execution block declaration (i.e. INDEPENDENT DO) a label is specified. This label is used later to synchronise the independent execution stream (i.e. the `doSomethingIndependently` block) with the

main execution stream (i.e. `doSomethingElse`). Serial execution continues from line 11 onwards.

Secondly, we consider this type of processing when executing two transactions that belong to the same request / evaluation plan (i.e. explicit inter-transaction concurrency).

```

20 {
21     ... // consider a University schema
22     INDEPENDENT DO TRANSACTION tr1
23         stNumb = LectureC.countStudents ( )
24         WHERE ( course.cNumb == "157.*" );
25         tr1.commit ( );
26     ENDDO;
27
28     DO TRANSACTION tr2
29         FOR EACH CourseC AS x {
30             selectedPapers.add ( x
31                 WHERE ( x.cNumb == "157.*" ) );
32         }
33         tr2.commit ( );
34     ENDDO;
35     ...
36 }

```

Transactions `tr1` and `tr2` are executed concurrently. No synchronisation command has been specified. Thus, both execution streams are only synchronised at the end of the corresponding block, i.e. line 36.

Finally, we will utilise the repetitive simultaneous execution of a block of independent statements on a common data set. In this example, we intend to process all statements of one iteration in a `for each` loop simultaneously with all statements of the next iteration etc.

```

40 ...
41 FOR EACH StudentC AS x INDEPENDENT DO
42     doSomething;
43 ENDDO;
44 ...

```

The `doSomething`-block is invoked independently for each object in class `StudentC`. Thus, we could also write:

```

50 ...
51 LABEL i1: INDEPENDENT DO
52     x = StudentC.first ( );
53     doSomething;
54 ENDDO;
55
56 LABEL i2: INDEPENDENT DO
57     x = StudentC.next ( );
58     doSomething;
59 ENDDO;
60 ...
61

```

```

62 LABEL in: INDEPENDENT DO
63     x = StudentC.last ( );
64     doSomething;
65 ENDDO;
66
67 WAIT i1, i2, i3, ..., in;
68 ...

```

On the other hand, it can be specified that a block of statements is executed concurrently while preserving the indicated ordering. This type of processing is particularly useful when processing collections. Let us consider an example:

Assume, we have a database that keeps track of student enrolments. Students have to apply for course enrolments. They will be approved into a particular course only if they meet all course pre-requisites. Before the beginning of a new semester we like to execute a routine that automatically approves applications which meet all respective course pre-requisites. Subsequently, we have to contact all students whose applications could not be approved automatically.

This procedure can be done in two subsequent steps. Alternatively, we could utilise concurrency (to be more precise, pipelining). This may be achieved as follows:

```

ForEach student Do in parallel
    Automatically approve all course
    applications.
Then
    Compile a list of students which have
    at least one unapproved application.
EndDo

```

Obviously, it is vital that the execution ordering is preserved.

In `iDBPQL`, we support this type of execution. The general syntax for such a specification is based on the `FOR EACH` loop statement:

```

"FOR EACH", Expression, "CONCURRENT",
"DO", Statements, {
    "THEN", "DO", Statements, "ENDDO", ';'
}, "ENDDO", ';'

```

The evaluation of the expression results in a collection value. Members of this collection are made available to the `DO`-statement first. Once those values have been processed they are pipelined to the `THEN DO`-statement. Thus, both statements may execute simultaneous while preserving execution ordering.

Let us consider some examples. First, we will call two methods on a collection of objects. The second method may be invoked on each object on which the first method has been executed successfully, i.e. objects are pipelined from the first invocation statement to the second invocation statement.

```

70 ...
71 FOR EACH myCollection CONCURRENT DO
72   // sorts all collection members
73   myCollection.sort ( );
74   THEN DO
75     // based on a sorted collection,
76     // duplicates are eliminated by
77     // considering 'neighbouring'
78     // collection members
79     myCollection.rmvDuplicatesSorted ( );
80   ENDDO;
81 ENDDO;
82 ...

```

Line 71 indicates the start of a block of statements that is to be executed concurrently. The two statements in lines 73 and 79 operate on the same collections and, thus, may execute concurrently as long as the execution order is preserved per collection object. So, every object that the `myCollection.sort ()`; statement releases (i.e. adds to its associated result queue) is passed on (i.e. pipelined) to the `myCollection.eliminateDuplicates ()`; statement. As a result, the execution of both statements may overlap (in a controlled manner).

Another means of specifying order-preserving, concurrent execution is within a `for`-loop. Within a `for each`-loop the pipelined object is selected explicitly. In addition to the `CONCURRENT DO`, we also have an `INDEPENDENT DO` in this loop.

```

90 ...
91 FOR EACH mydb.SalaryC AS x
92 CONCURRENT DO
93   x.addBonus ( 2,000 );
94   THEN DO
95
96     Label i1: INDEPENDENT DO
97       x.printPaymentSlip ( );
98     ENDDO;
99
100     extMail.add ( x ) WHERE (
101       x.hasInHouseMailAddress ( )
102       == FALSE );
103
104     WAIT i1;
105
106   ENDDO;
107 THEN DO
108   x.salaryProcessed (
109     date.today ( ), time.now ( ) );
110 ENDDO;
111 ENDDO;
112 ...

```

For each object `x` in the collection-class `mydb.SalaryC`, the `addBonus` method is invoked. Sub-

sequently, this object is pipelined to both the execution unit invoking the `printPaymentSlip` method as well as to the execution unit that maintains a set of salary objects that do not satisfy the boolean `hasInHouseMailAddress` method. Once, both invocations have been completed (refer to the `WAIT` in line 103), we can pipeline the object `x` to the fourth execution unit invoking the `salaryProcessed` method.

While both order-preserving examples outlined above only refer to operations within a single transaction, the same concept can be applied across transactions. Let us demonstrate this next:

```

120 ...
121 DO TRANSACTION tr1
122   salCol = mydb.SalaryC;
123
124   DO TRANSACTION tr2
125     FOR EACH salCol AS salObj
126     CONCURRENT DO TRANSACTION tr1
127       salObj.addBonus ( 2,000 );
128
129     THEN DO TRANSACTION tr2
130       // implies ordering: tr1 first
131       // and then tr2
132       salObj.printPaymentSlip ( );
133     ENDDO;
134   ENDDO;
135 ENDDO;
136 ENDDO;
137 ...

```

Here, transaction `tr1` obtains the `SalaryC` collection and adds all bonifications. After each bonus is added, the objects is handed over to transaction `tr2`, which prints all payment slips.

4.3. Implications

Supporting explicit concurrency has implications on other components of the DBS and on the syntax of `iDBPQL` itself. We will look at the different types of simultaneous processing and outline their implications:

- Firstly, there is the `INDEPENDENT DO` statement that applies to sequences of `iDBPQL` statements inside a single transaction or, in the event that no transaction is defined, the entire evaluation plan. Since access to shared data must be from within a transaction, the latter case can be neglected since its evaluation will not involve the transaction management system (only local, non-shared data is accessed). Considering the former case, the simultaneous execution only applies to operations within the same transaction. This, however, will not have implications for other DBS components,

in particular the transaction management system. According to the ACID principle, data consistency has to be preserved by each transaction, when run in isolation, and the programmer is responsible for ensuring this property.

- Secondly, there is the `INDEPENDENT DO` statement that results in the simultaneous execution of multiple transactions originating from the same evaluation plan, i.e. same main execution stream. From the transaction management system's point of view, these transactions are serialised as any other concurrent transactions (i.e. those discussed in Section 4.1). However, some syntactical constraints must be observed when using this type of explicit concurrency. Multi-threaded transactions must commit or abort before rejoining the main evaluation plan and synchronisation commands (i.e. `WAIT` statements) must not form cyclic waiting conditions.
- Thirdly, there is the `CONCURRENT DO` statement that applies to operations on collections either across transactions, within a single transaction or, in case no transaction is defined, the entire evaluation plan. Operations are synchronised implicitly by pipelining objects that the first operation has released to the second operation etc. Thus, concurrent access to the same collection object is enabled while access to the collection members is synchronised. This form of cooperation is different from the modes of processing traditional DBSs usually perform. As a consequence, the transaction management system must not be only able to distinguish serial and independent (i.e. of operations allowing shared access or operations operating on unrelated objects) execution of operations within the same transaction but also the coordinated execution of possibly conflicting operations on the same collection. Corresponding extensions to transaction models and correctness criteria are proposed in [1]. The suggested transaction model distinguishes between two partial orderings, which are weak order (i.e. data flow takes place through DBS objects) and strong order (i.e. external flow of information between operations or transactions). While the former enables simultaneous processing, the latter implies serial execution. The corresponding correctness criteria, stack-conflict consistency, permits parallel execution of weakly ordered operations given that their serialisation graph is preserved.

While the support of explicit simultaneous execution requires a more sophisticated transaction management system, it offers the potential to significantly increase system performance. For instance, assume that we have two subse-

quent operations accessing the same collection. This collection may be of a size that does not fit into the available main memory. Serial evaluation is likely to result in two consecutive scans, which degrades system performance. On the contrary, the `CONCURRENT DO` block enables the evaluation of both operations with one scan.

5. Conclusion

In this paper, we presented features, which relate to simultaneous processing, of a newly developed integrated database programming and querying language (i)DBPQL. Main focus was on the different ways simultaneous processing is supported explicitly and implicitly, and also on their implications on other database system components such as the transaction management system.

References

- [1] G. Alonso, S. Blott, A. Feßler, and H.-J. Schek. Correctness and parallelism in composite systems. In *Proceedings of the 16th ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 197–208. ACM Press, 1997.
- [2] M. P. Atkinson and P. Buneman. Types and persistence in database programming languages. *ACM Computing Surveys (CSUR)*, 19(2):105–170, 1987.
- [3] F. Bancilhon, T. Briggs, S. Khoshafian, and P. Valduriez. FAD, a powerful and simple database language. In P. M. Stocker, W. Kent, and P. Hammersley, editors, *Proceedings of 13th International Conference on Very Large Data Bases*, pages 97–105. Morgan Kaufmann, 1987.
- [4] S. Feuerstein and B. Pribyl. *Oracle PL/SQL Programming*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2002.
- [5] B. E. Hart, S. Danforth, and P. Valduriez. Parallelizing a database programming language. In *Proceedings of the 1st international symposium on Databases in parallel and distributed systems*, pages 72–79. IEEE Computer Society Press, 1988.
- [6] M. Kirchberg. An overview of the object-oriented database programming language DBPQL. In J. Cardoso, J. Cordeiro, and J. Filipe, editors, *Proceedings of the 9th International Conference on Enterprise Information Systems (ICEIS)*, volume 1. INSTICC Press, 2007.
- [7] M. Kirchberg, K.-D. Schewe, A. Tretiakov, and B. R. Wang. A multi-level architecture for distributed object bases. *Data & Knowledge Engineering*, 60(1):150–184, January 2007.
- [8] K. Subieta. LOQIS: The object-oriented database programming system. *Lecture Notes in Computer Science*, 504:403–421, 1991.