APPLICATION NOTE

LOCKING IN POLYHEDRA®
How a database system can offer a locking mechanism appropriate for real-time use

Abstract

This document looks at the Polyhedra transactional model including its ability to add database locks, and shows how these capabilities can be used in real-time applications.

Contents

ENEA

An overview of Polyhedra and locking

An overview of Polyhedra®

Polyhedra is a family of relational database management systems designed from the start for use in embedded applications. It has two main flavours: Polyhedra IMDB and Polyhedra FlashDBMS, with the former being available for 32-bit platforms (Polyhedra32, the ŏoriginalô version) and 64-bit platforms (Polyhedra64). The difference between these products is that Polyhedra IMDB is in-memory for speed, and has journalling and fault-tolerant mechanisms to ensure data persistence and system availability, whereas Polyhedra FlashDBMS trades performance against RAM footprint by using a tuneable in-memory cache in front of a file-based database (assumed to be on a flash-based file system). In 2012 a new product was released: Polyhedra Lite, a free version (subject to license conditions) of Polyhedra32 IMDB, but omitting some functionality such as support of fault-tolerant configurations.

All Polyhedra products support an extended SQL-92 subset and the ODBC and JDBC client libraries conform to the international standards. They have object-oriented features, such as table inheritance (which simplifies and speeds up many queries and updates) and behaviour (to perform knock-on actions and additional integrity checks). Polyhedra was designed from the start for client-server use, with full interoperability between 32-bit and 64-bit versions of Polyhedra. When all the software is running on the same machine, the client-server architecture has three main benefits:

- The software naturally allows for multiple clients, which could be running the same application software (on behalf of different users, say) or performing different parts of the overall application.
- Where supported by the operating system and hardware platform, the client application(s) and the server will run in separate address spaces, protected from each other - so a failure of an application will not damage the integrity of the database, nor stop other parts of the application running. Also, on an SMP multicore or multiprocessor machine, clients can be running on separate cores from each other and from the server, and thus allow the overall application to make more use of the available CPU cycles.
- The client-server architecture naturally extends to allowing client and server to be on separate machines, to allow the application to be distributed. When clients are remote, Polyhedra automatically handles issues such as differences in endianism.

Polyhedra servers can operate stand-alone, with each instance handling a separate database, and in addition both Polyhedra IMDB and Polyhedra FlashDBMS can operate in master-standby mode, where one server is acting as a hot standby of another server and has a read-only copy of the database. Polyhedra is fully transactional, and satisfies the Atomic, Consistent and Isolated properties needed for ACID compliance (see sidebar). Polyhedra FlashDBMS transactions are Durable, but in Polyhedra IMDB durability can be balanced against performance: critical changes to the data are preserved by streaming journal records to a log file, and client applications can choose whether the success of a transaction is to be reported immediately or when the log file has been flushed.

"The ACID Test"

ACID is a commonly-used abbreviation for **A**tomic, **C**onsistent, **I**solated and **D**urable, relating to desirable properties of transactional data management systems:

- **Atomic –** Transactions are all-or-nothing: the system never does just part of what is asked.
- **Consistent –** Transactions take the system from one consistent state to another: the system rejects any transactions that break the rules.
- **Isolated –** Transactions operate independently, as though fully serialized, with intermediate, mid-transaction states hidden.
- **Durable –** If a database indicates that a transaction has successfully completed, the application can rely on the changes being preserved - even through a variety of system failure scenarios

Implicit in the above is that the ONLY way of reading or writing to the data is via transactions. All of these transactional properties are vital for a reliable data storage system. Most database systems will offer ACID-compliance, though this is much less common in simpler data stores.

Polyhedra has a special feature called ŏactive queriesô that allows clients to keep up to date without polling. Basically, for the lifetime of the query the server remembers enough about the query (and the previously-transmitted result set) to know when it has become out of date ï so when the transaction completes it can send a ŏdeltaô to the client that launched the query to bring it up to date. There is a ŏdelta mergingô mechanism to avoid problems if the client is slow.

A sample database

In this paper we shall present various examples, all of which will assume we are using a database modelling a registry of births, deaths and marriages. Also, we shall assume that all marriages are monogamous ï you cannot be married to two people at the same time. (This constraint might not hold where you live, but this is just a simple example for the purposes of exposition.) The schema for the database can be defined by the following SQL:

```
create table person

    (   persistent
    ,   id          integer primary key
    ,   name        large varchar not null
    ,   born        datetime
    ,   died        datetime
    ,   ismale      bool not null
    ,   birthplace large varchar
    );

create table marriage

    (   persistent
    ,   id        integer primary key
    ,   spouse1   integer not null references person
    ,   spouse2   integer not null references person
    ,   started   datetime not null
    ,   location large varchar
    ,   ended     datetime
    ,   whyended large varchar
    );
```

This simple schema ignores issues such as parentage. The `died` and `ended` fields are used to record, respectively, the date a person died and when a marriage ended ï so null values will be assumed to mean the person is alive, and the marriage on-going.

Polyhedra Locks - an overview

In principle, database locking allows an application to claim parts of the database, so that others cannot make changes while the client with the lock is deciding on the changes it wants to make. Earlier versions of Polyhedra did not allow locking, providing instead a way of attaching trigger code that can automatically perform sophisticated integrity checks (and then either abort ŏbadô transactions, or perform corrections and knock-on actions as part of the transaction). However, there are circumstances where users neither want to attach trigger code nor want to adopt an in-house convention to implement an application-specific locking mechanism, so we have now added locking capabilities to Polyhedra ï but we have been careful to do this in a way that is appropriate to the needs of real-time applications.

Technically, the simplest way of providing locking is to allow the client to grab the whole database ï but this is too heavy-handed an approach for most purposes, so many database systems allow clients to be more selective in their locking, down to the level of individual tables or even individual records in a table. Polyhedra goes one better than that, so that it is even possible to lock individual fields of a record.

This level of granularity is critical for real-time systems ï for example, if you have a table holding information about analogue sensors, we need to allow one client to lock the configuration fields for a record without stopping other applications updating the ŏcurrent valueô fields as soon as new readings become available.

Polyhedra allows locks to be specified as pessimistic or optimistic, which controls which transaction fails if someone tries to alter a locked piece of data: a pessimistic lock will stop others altering data you have locked, whereas an optimistic lock will cause your transaction to fail if somebody else alters something you have locked. To avoid someone locking up the database for some time, the database administrator can set up a timeout for pessimistic locks: after this time, pessimistic locks will automatically be converted to optimistic locks. As users cannot place locks that conflict with existing pessimistic locks, and the conflicting lock request will fail rather than waiting for the other lock to be freed, all possibilities of deadlocks or deadly embraces are avoided.

The new locking mechanism works alongside any trigger code that may have been attached to the database, so the locks do not just monitor changes directly requested by other clients, but will also protect locked data from being indirectly changed by code that has been triggered by other alterations to the database. And, of course, the new locking mechanism is integrated into Polyhedraǒs user-based security system, so the database administrator can control which tables (and which attributes of a table) someone can lock.

Placing locks

When Polyhedra was extended to add a locking mechanism, it was important to ensure existing applications were unaffected. Thus, users have to explicitly ask for locks to be placed, using extra syntax in the SELECT statement. For example, to place a pessimistic lock on particular records of a table, you could say

```
select id, name, born, birthplace, died
       from person where name = 'Hugh' or name = 'Anne'
       for pessimistic update
```

This retrieves a number of rows from the `person` table, and stops anyone from altering them for a while. More accurately, it stops anyone from altering the fields that have been retrieved ï but other fields in the record can be altered, and it is even possible to create another record that satisfies the query without affecting the lock. The query does not stop anyone deleting a locked row ï if that behaviour is wanted, it can be achieved by adding **`or delete`** to the end of the query.

If you place an optimistic lock, you can also specify that you want to know if anyone makes a change that affects which records satisfy the `where` clause:

```
select id, spouse1, spouse2, ended
       from marriage where (spouse1=13 or spouse2=13 or spouse1=100 or spouse2=100)
                    and ended is null
       for condition or update or delete
```

This retrieves all marriage records relating to two particular people. If someone changes anything in the database that would affect the returned fields, or does anything that affects what records would have been affected by the query, then the lock is triggered. As it is an optimistic lock, this means that your next transaction would fail, rather than the transaction that changed the `marriage` table.

Locks normally last until your next transaction, so the effect of the followingé

```
insert into marriage (id, spouse1, spouse2, started,              location)
              values (9,  13,      100,     date ('15-May-1986'), 'Cardiff');
```

é would be to clear both of the locks placed earlier, and ï assuming no-one had done anything to affect the optimistic lock ï create a record in the marriage table. Locks are also cleared if you issue a rollback, or close a connection; in addition, locks are automatically cancelled in fault-tolerant configurations if a failover occurs.

More details on Polyhedraĝs locks

The core transactional model in Polyhedra

Polyhedra is designed for real-time use, and therefore the emphasis is on speed and efficiency. To this end, the transactional model is deliberately kept very simple. In essence:

- A client application sends in a request, which is put on a FIFO queue.
- Items were taken off the queue, and processed sequentially. Once the request is completed, the database engine prepares the response for the client, and moves on to the next item on the queue. The response is sent asynchronously to the client.
- A request can be
  a) a single query (which can be a static query or an active one);
  b) a set of DML statements, such as insert, update and delete;
  c) a schema change request;
  d) a set of DML statements combined with a single static query; or,
  e) an administrative request (such a SHUTDOWN or SAVE command).

(The DML+SELECT request is provided to allow a way for a client to make a sequence of changes, and then inspect the database before any other transaction has had a chance to modify the database further.)

In fact, it is a slight over-simplification to say Polyhedra uses a FIFO queue for transactions, as the database engine uses an internal system of semaphores, non-exclusive table-level read locks and exclusive table-level read-write locks to allow queries to be performed for some clients while another client is altering other tables. Logically, the semantics remain that of full serialisation via a FIFO queue, but the internal, table-level locks allow for some overall performance improvements when using Polyhedra on multicore machines.

CL triggers

Polyhedra automatically enforces entity integrity (every table must have a primary key and the column or columns chosen to be the primary key should be unique and not null), domain integrity (each data value must be the appropriate type for the column) and referential integrity (any field in a table that is declared a foreign key can contain either a null value, or only values from the referenced table's primary key). However, most databases need to enforce additional constraints ï for example, in the marriage example described earlier, you might want to ensure that you donĝt create a new marriage record if either party has been recorded as dead, and that a person cannot be married to more than one person at a time. (If the database allows same-sex marriages, you should also ensure nobody is married to themselves!) Such application-level constraints can be enforced by each application performing appropriate checks before updating the database, but Polyhedra allows trigger code (written in a proprietary high-level language called CL) to be attached to the database to perform such checks. This simplifies the job of the application writers (as they donĝt need to perform the checks or any knock-on actions), increases the integrity of the database (as the checks cannot be forgotten or evaded), and also improves the overall efficiency of the system (as there is less interaction between the client and the server). The following CL code sample checks that the two parties of a marriage are both of the correct gender, and were flagged as alive when the marriage record is created:

```
script marriage

    on  create
        local string str = "herself"
        if started > now () then
            abort transaction "start date cannot be in the future."
        else if exists ended then
            if ended > now () then abort transaction "end date cannot be in the future."
            -- the ended field is not null, and in the past - so the marriage is over,
            -- and this record does not affect the validity of future marriages.
            -- Consequently, we can ignore the record.
            exit
        else if spouse1=spouse2 then
            if ismale of spouse1 the set str to "himself"
            abort transaction name of spouse1 && "cannot marry" && str
        end if
        -- many more checks could be added here, to ensure the new marriage conforms
        -- with the laws or conventions applicable in your country! For example, if
        -- same-sex marriages are illegal, you could check if ismale of spouse1 equals
        -- ismale of spouse2. The following rule would apply in most juristictions:
        if exists died of husband or exists died of wife then
            abort transaction "both spouses must be alive to marry."
        end if
    end create

end script
```

(It would be simple to add more code to ensure certain fields cannot be changed after the record has been created, and ï with a small bit of denormalisation so that the record for a person has a reference to the current marriage, if ï it would be straightforward to ensure a person cannot be married to two people at once, and that when a person is flagged as dead the marriage is marked as ended. A trigger on the `ended` field of the `marriage` table could in turn clear the references to a marriage that has finished from the relevant records in `person`.)

The above CL code is performed as part of the transaction that performed the insert, and any changes made to the database by CL code can trigger further CL methods, which is again performed in-line as part of the transaction. Thus, by the time the transaction is complete all the associated CL trigger code will have been run to completion[1], with the only interaction with the client being an indication as to whether the transaction failed or succeeded.

Optimistic concurrency via active queries

As well as monitoring for changes, Polyhedraôs active query mechanism can be used to update the database. Basically, the client updates its local copy of the result set, and this is interpreted as a request for the corresponding change to be made to the database. Clearly, this is only allowed if the query is an updateable one ï no joins, for example. When the change request reaches the top of the queue, the database engine checks if there has been any conflicting changes (such as the record to be altered being deleted, for example, or another change has already been made to a field that we are trying to alter) ï if so, the change request is automatically cancelled and the client will get a delta to reflect the current state. Thus, when updating through an active query (or a number of active queries, if the changes are submitted as a batch) you get a form of optimistic locking: your transaction fails if it clashes with a change made by someone else. Your transaction is permitted if someone has changed fields that you are not changing: while this is sometimes an advantage, it can theoretically lead to problems if, say, you are using the values of two fields to determine a suitable new value for a third field in a record.

[1] (This is a slight over-simplification, as the CL programmer can choose to ósuspendô the code, for resumption at a later time in a new, private transaction. See the CL reference manual for more details.)

Extending the transactional model with locks

Starting with release 8.7 (March 2013), Polyhedra has been enhanced with a locking mechanism. This complements the core transactional model, by providing a way for a client application to query the database and then update it, with confidence that no-one else has made conflicting modifications in the meantime. You can either consider it a way to avoid having to attach CL code to the database to perform integrity checks ï or a way of performing updates with supplementary checks over and above those automatically enforced by the database engine and any CL that may be attached to the database.

Three major considerations in the design of the locking mechanism were backwards compatibility (so that older applications that did not use locking have exactly the same semantics); efficiency; and, suitability for real-time applications. Thus, one essential feature was that locks should be fine-grained: it should be possible to lock the configuration fields of a record containing information about a sensor, while still allowing the current value field to be updated as new readings become available.

Another requirement, stemming this time for the need for backwards compatibility, was that lock requests would have to be explicit: this was achieved by extending the syntax of the SQL SELECT statement so that users can request a lock. The new clause has the syntax

```
[ for [ <lock-mode> ] <lock-operation> [ { or <lock-operation> ...} ] [ without fetch ] ]
```

The `without fetch` clause is an instruction that the lock should be placed, but without returning the result set to the client. This allows a cheap way of placing a lock on a whole table, without having to send a copy of the table to the client.

The lock mode can be `optimistic` or `pessimistic`, (with the default being `optimistic`) and the lock operations are `insert`, `update`, `delete` or `condition`. We shall now look in detail at what these mean.

Lock types

A lock is a way of stopping two transactions making conflicting transactions. Consider, for example, a case where one client queries the database and then updates the database using information derived from the earlier query. If, though, another client has come along in the meantime and changed the information the first client looked at, the update done by the first client would be based on out of date information (and thus probably wrong). To prevent this we need to prevent one or other of the transactions taking place ï or delay the transaction done by the second client until the first client has completed both steps.

A traditional approach would be for the first client to place a lock on the data it queried, and for the second clientõs update to be delayed until the first client has completed its transaction. The problems with this approach are twofold: the second client could be held up for some time, and you can get deadlocks. Polyhedra avoids these problems by going for no-wait locks (where a lock request fails if it conflicts with another lock), and providing users with a way of choosing which of the two transactions would fail. Thus, if a change made by another client causes a lock conflict, then its transaction would fail if the lock was a pessimistic one, whereas if the lock is optimistic the transaction that fails is the one for the client that placed the lock (assuming, of course, that the other transaction ran to completion).

In general, it is better to use optimistic locks, as then it does not matter how long the client that placed the lock waits before performing the transaction (or rolling it back): it does not stop other clients updating the database. Pessimistic locks, on the other hand, can cause problems especially in real-time applications ï so Polyhedra allows the database administrator to set up a time-out, after which a pessimistic lock automatically converts to an optimistic one. This time-out can be short, as it is specified in milliseconds!

Lock operations

As mentioned above, there are 4 lock operations: `insert`, `update`, `delete` and `condition`, and these can be used in combination. The semantics are:

insert	An insert lock is triggered if another transaction inserts a record into the table being queried (or into any table derived from this table). It does not matter if the new record satisfies the `where` clause of the query that established the lock; any insert is caught.
update	An update lock is triggered if anyone updates a field that appears in the result set of the query that established the lock (regardless of whether `without fetch` had been specified). If there was a `where` clause, then altering a record that is not in the result set does not affect the lock, nor does altering a field that has not been requested by the query. (In the absence of a `where` clause, an update lock is also affected by alterations to records that were not present when the lock was created.)
delete	A delete lock is triggered if anyone deletes a record that appears in the result set of the query that established the lock (regardless of whether `without fetch` had been specified). If there was a `where` clause in the query that established the lock, deleting a record that is not in the result set does not affect the lock. (In the absence of a `where` clause, a delete lock is also affected by the deletion of records that were not present when the lock was created.)
condition	A condition lock is triggered if a changed is made that would affect which rows were returned by the query that established the lock. So, assuming the query had a `where` clause, if a new record is inserted that matches the `where` clause, or a change is made that alters whether a row matched the `where` clause, the lock is triggered. (A condition lock is not triggered by the deletion of a record from the result set unless it has also been specified to be a delete lock.) At present, only optimistic condition locks are supported.

(Note that it does not matter if the operation that triggers a lock was directly requested by a client, or if it is the result of, say, some CL code attached the database, perhaps to a different table. So if a client inserts a record in one table, and an `on create` handler on that table creates a separate record in a table that happens to be monitored by an insert lock, the lock is triggered.)

To clarify the above, let us look at see some sample lock queries, and see how they would be affected by some updates. First the locking queries:

```
select id, name, born, died from person …
```

1
```
where name = 'Hugh' or name = 'Anne' for pessimistic update
```
2
```
where name = 'Hugh' or name = 'Anne' for optimistic update
```
3
```
where name = 'Hugh' or name = 'Anne' for pessimistic insert or delete
```
4
```
where name = 'Hugh' or name = 'Anne' for optimistic condition or update
```
5
```
where name = 'Hugh' or name = 'Anne' for optimistic condition or update or delete
```
6
```
where name = 'Hugh' or name = 'Anne' for optimistic condition or update or insert
```
7
```
for pessimistic insert or update or delete without fetch
```

(Due to the presence of the `without fetch` subclause, query 7 does not actually return any records to the client, though notionally all records in the table are in the result set.) Let us assume that there is one record with the name `Hugh`, another with the name `Anne`, and a third with the name `Fred`. The updateé

```
insert into person (id, name, male) values (007, 'James', true)
```

é would fail if lock 3 or 7 is in place, and would kill the lock transaction that placed lock 6; it does not matter that the new record does not match the `where` clause of lock 3 or 6, as any insert triggers the lock. Locks 1, 2, 4 and 5 would be unaffected as they do not set up insert locks. By comparison,

```
insert into person (id, name, male) values (008, 'Hugh', true)
```

é would fail if lock 3 or 7 is in place as before, but this time would kill the lock transaction that had placed lock 4, 5 or 6 as the new record satisfies the original query. Locks 1 and 2 would be unaffected, despite the fact that the new record satisfies their `where` clauses.

```
update person set birthplace = 'Swansea' where name = 'Hugh'
```

é would not be affected by any of the locks given above, since none of them specify the column being updated. If, instead the operation had been `set died=now() where name='Hugh'` then the transaction would have failed if lock 1 or 7 were in place, and would kill the lock transaction that had placed lock 2, 4, 5 or 6. Moving on to deletions,

```
delete from person where name = 'Fred'
```

é would fail if lock 7 is in place, but locks 1-6 would be unaffected as the record being altered is not in their result sets; in fact, only locks 3, 5 and 7 can be affected by a deletion, as only they explicitly specify a delete lock. If, however, we tried to delete the record with name `Hugh`, the transaction would fail if lock 3 or 7 is in place, and would kill the lock transaction that had placed lock 5.

A `for optimistic condition or update or delete` lock is probably the most useful type of lock, as it is triggered whenever someone else does something that means your original query is likely to be out of date, and it unaffected by other changes. For this reason, we recommend its use in preference to other locks, where this makes sense in your application.

Creating and releasing locks

As has been shown above, locks are created by means of special queries. When a lock is established, the client can be said to be running a ölock transactionô. It is possible to add further locks, which can be any mixture of optimistic and pessimistic locks, with the lock transaction normally being terminated by a request for some changes. It is possible to cancel the lock transaction, but the way that is done varies according to the API that is being used. For SQLC, for example, you would issue the `ROLLBACK` command, which sends a special message over the client-server connection. (`ROLLBACK` is just a command that is understood by SQLC; as with `COMMIT`, it is not part of the SQL syntax recognised by the Polyhedra SQL engine.) For ODBC, you would use SQLEndTran with the SQL_ROLLBACK completion code.

For a lock to be accepted, it must not conflict with other pessimistic locks that have been placed by other clients. Conflicts can only happen in the case of pessimistic locks, when any of the following apply:

- Both use an INSERT lock operation and either they both lock the same table or one of the tables is derived from the other.
- Both use an UPDATE lock operation and the fields they lock intersect.
- Both use a DELETE lock operation and the rows they lock intersect.

If a lock is rejected because of a conflict, the lock transaction is cancelled, along with all the locks that form part of that transaction; the client does not have to roll back the transaction. The lock transaction also fails (with the immediate cancellation of all its locks) if any of the optimistic locks have been triggered; the client will discover this when it attempts to place another lock (of any type) or when it tries to complete the transaction.

If a connection has been set up has as fault-tolerant and a fail-over occurs, any locks that were set up prior to the failover will have been lost.

Locking and Security

Polyhedra supports a user-based security policy, where individuals can be granted access rights at the table and column level. You could, for example, make a table publicly available, and also allow anyone to alter certain columns ï but only allow specific people the right to alter the other fields, or to insert or delete records in the table. This has been extended to allow the database administrator to control who can place pessimistic locks on a table, down to the column level, so you can ensure that only trusted people can lock the critical tables. (Anyone can place an optimistic lock, provided they are entitled to read the information ï but as this does not block other people from updating the information öprotectedô by the optimistic lock, there is no danger in this.)

Further Details

More information about Polyhedraʼs locking mechanisms can be found in the Polyhedra SQL reference manual. In the case of any conflict between what is given in this document and the most recent release of the Polyhedra reference manuals, the reference manuals should be considered authorative.

Summary

The new locking mechanism in Polyhedra adds flexibility to the way Polyhedra applications can be put together to ensure a high degree of consistency and integrity in the information held within the database. At the same time, the fine granularity of the locks combined with the ability to add both optimistic and pessimistic locks, and also the ability to control both the duration of pessimistic locks and their placement, mean that Polyhedraʼs suitability for use in real-time applications has been maintained.

For more details of the Polyhedra® product family, please visit our web sites, www.polyhedra.com and developer.polyhedra.com, or email us at info@enea.com. For information about Enea, visit www.enea.com