External Database Extension Framework

Alexander Adam[1] and Wolfgang Benn[2]
[1]*Dimensio Informatics GmbH, Brückenstraße 4, 09111 Chemnitz, Germany*
[2]*Chemnitz University of Technology, Straße der Nationen 62, 09111 Chemnitz, Germany*

Keywords: Database Extension, External Indexing, Query Processing, Legacy Systems.

Abstract: Database systems nowadays show an incredible amount of extensibility interfaces, ranging from simple user defined functions to extensible indexing frameworks as seen in e. g. DB2 and Oracle. Approaching new projects using these interfaces definitely is a valuable option. For legacy systems, already set up and running in production environments, these options are often not available since most of them impose a change in the applications. In this work we present a database extension framework, that enables the user to define functionality which does not reside inside the database. We show different ways to integrate it into existing application landscapes without further modifications.

SCIENCE AND TECHNOLOGY PUBLICATIONS

1 INTRODUCTION

Currently there is a vast amount of database systems on the market, ranging from small embedded solutions up to full scale cluster versions. To meet the needs of most customers, vendors integrate interfaces into their systems to extend the database systems capabilities. They range from user defined functions and types up to user defined indexing and data storage. (Stolze and Steinbach, 2003; IBM, 2009; Rich, 2011; Belden et al., 2009)

For new projects all of these options are available for usage and should be thoroughly evaluated. However, running systems in production environments limit the amount of possibilities to modify the databases structure. The main reason for this limitation is the need to adapt the queries the application passes to the database system. In deployed environments changing the application can be regarded as not feasible. This leaves only to alter non-query-changing options in the database.

Database administrators who are responsible for keeping their systems up and running face this situation. As the amount of data stored grows, systems change their behavior: certain content tables begin having increasing data retrieval times, database partitions reach their size limits and many more. All of these effects may lead to a degrading application performance. Common approaches inside the database are (not limited to) the use of partitioned tables, indexes and the optimization of the database buffer pools. On the hardware side there is the KIWI approach (“Kill It With Iron”). That is more main memory, more computing units, faster storage systems, clustering and in-memory solutions.

As shown in (Leuoth et al., 2010) not all situations can be resolved by applying the aforementioned standard methods changes inside the database would be necessary. In this paper we present an extensibility framework for, especially but not limited to, relational database management systems (RDBMS) to master the challenge of introducing new functionality virtually “into” the database system. It can be applied in production systems, where neither the database system nor the software are allowed to be changed to suite new needs or performance requirements.

The following sections present the current approaches to these problems and their limitations. As examplary scenario used throughout this paper the integration of an indexing method was chosen. After that we analyze the behavior of an environment containing an RDBMS and how our framework can be transparently plugged into it. Finally we present an implementation of a database external indexing and result cache.

Adam A. and Benn W..
External Database Extension Framework.
DOI: 10.5220/0004896302490255
In *Proceedings of the 16th International Conference on Enterprise Information Systems* (ICEIS-2014), pages 249-255
ISBN: 978-989-758-027-7
Copyright © 2014 SCITEPRESS (Science and Technology Publications, Lda.)

2 CURRENT APPROACHES

2.1 Scenario

Sticking to the example of indexing, RDBMS currently include many indexing methods. In the field of high dimensional indexing (50+ dimensions) there is no efficient, non-KIWI solution available. There is an indexing approach for exactly this problem, developed at the Chemnitz University of Technology, the ICIx (Görlitz, 2005). This multi-dimensional index resides outside of the database system and has to be used explicitly by an application using an API. It delivers primary keys of the relevant data to the application which in turn includes these information to enrich its queries to the database.

We now have a look at how current RDBMS can help to integrate such an index.

2.2 User Defined Functions

A first approach might be to work with user defined functions (UDF). User defined functions are functions made available to the user through the SQL language interface. They are either written in the database systems internal SQL programming language (e. g. PL/pgSQL (Pos, 2009), PL/SQL (Rich, 2011) or SQL/PSM (IBM, 2009)) or using an external language that can be interfaced on a binary level with the RDBMS.

Using UDF to incorporate a call to an external index, one has to define a function that calls the API of the external index. We will not cover the details of how to call an external API function, just assume, that it is possible. Finally the new UDF has to be used in the queries, so that it is called.

```
SELECT *
FROM   employees
WHERE  id IN ( idx_lookup_smaller(
         employees.salary, 2000)
       )
```

Obviously every query, using the index, has to incorporate the UDF. Every data modification, has to be transported manually to the index, e. g. using another UDF. All in all that requires an adaption of the application and the integration of the new functions into the RDBMS. We consider the first point being the more pressing one, as it requires the software vendor to change the software. Modifications to the database can be isolated in other namespaces and are thus very unlikely to interfere with the current operation.

2.3 Extensible Indexing Frameworks

Extensible indexing frameworks give the user a specialized interface, that enables the user to define functions which are called on certain events. It further gives a clean integration into the SQL language. The RDBMS guarantees transaction safety of the actions triggered in certain events. Those events include the creation of the index, modification of data and the selection on columns managed by the index. For all but the selection the extensible indexing frameworks provide a transparent access to the index. On selection a special predicate has to be defined and used, taking the form of a function thats return is compared to a value. (Belden et al., 2009) The following example illustrates the use of such an indexing extension:

```
SELECT *
FROM   employees
WHERE  idx_lookup_smaller(
         employees.salary, 2000)
       ) = 1
```

So the selection operation is not transparent to the application and thus the application has to be adapted. Though not as deeply integrated as the UDF approach, not many vendors will be reluctant to integrate a functionality on just a few customer requests.

2.4 Current External Approaches

Our example of the index integration is relatively complex. For less complex functionalities, approaches that are integrated into the network communication exist.

The TDS Protocol Analyzer (TDS, 2013) is a tool that operates on the TDS (Tabular Data Streams) protocol used to communicate with the Microsoft SQL Server. Its use is the interception, recording and analysis of packets sent by a client to the database. These packets can be inspected for statistical analysis and vulnerability inspections. (Guo and Wu, 2009)

GreenSQL is an open source proxy with the goal to increase the security of MySQL and PostgreSQL. In a commercial version it also supports Microsoft SQL Server with TDS. It scans the communication of clients with the database for suspicious requests that might be able to exploit security vulnerabilities. It also is able to work as a firewall. (Gre, 2013)

In a last project, the Security Testing Framework, syntactically wrong packages for the DRDA (Distributed Relational Database Architecture) (The Open Group, 2011) protocol are generated, to test the robustness of the implementation. (Aboelfotoh et al., 2009)

All of these tools concentrate on a top level view of the protocol and do not deeply integrate into the interaction of the application with a database.

3 DATABASE INTERACTION

To find integration points we first analyze how applications interact with databases. For the rest of this paper, we assume that an application is a process using a library to communicate with a database process. The communication channel is a network connection. This basic setup is also shown in Figure 1.

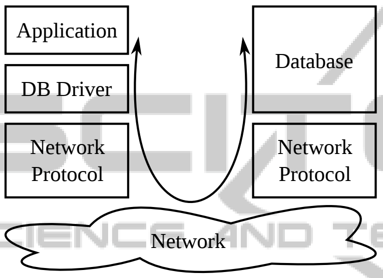

Figure 1: Basic scheme of how an application communicates with a database.

The way of a query, beginning from the application, includes the following steps:

1. application hands over query to RDBMS API
2. application gives additional information (such as bind parameters) to RDBMS API
3. RDBMS API sends network packet with query to RDBMS
4. RDBMS receives network packet, extracts information and processes them (intentionally shortened here)
5. RDBMS sends results over network to application's RDMBS API
6. RDBMS API receives and extracts network packet
7. RDBMS API provides results to application

We will have a closer look at the steps outside of the database system and application. These are the API communication (steps 1, 2 and 7) and the networking layer (steps 3 and 5). As the network level simply reflects the steps initiated at the API level, we can introduce an abstract interface for both of them. The abstraction layer we use is thereby composed of the above steps and named according to other event driven functions with the prefix `''on..."` and the suffix `''...request"` and `''...reply"` to indicate the communication direction. For our work we first implement the basic functions `prepare`, `bind`, `execute` and `fetch`. The use of these functions to implement helpful functionality, is shown in the following section and illustrated later in Figure 3.

Below we describe methods to integrate functionality into these layers.

3.1 API-level Integration

The first possibility, we have a closer look at, will be the integration into the API layer. Usually, database vendors deliver at least one library to interact with their database system. This library exposes a certain API. Examples for these vendor specific APIs are e. g. libpq(Pos, 2009) and OCI (Melnick, 2009). To have an abstraction layer on top of these, portable APIs were developed. Among these are ODBC (ODB, 1995), JDBC (Menon, 2005), OLE DB (Blakeley, 1997)and ADO.NET (Kansy and Schwichtenberg, 2012).

An application links against such a database library and subsequently uses its API. We showed that it is possible to redirect the API calls to an own library that implements the exact same interface as the library originally used by the application.

To minimize development efforts, one can use system specific support. In Linux there is the `LD_PRELOAD` environment variable, allowing a user to load a specific library before any other library. This library populates the symbol tables of the process. Any other library that is loaded afterwards is only able to set the symbols not already set. In Microsoft Windows one can generate a wrapper library, which routes all symbols, that are not directly needed to the original ones and only "overloads" the ones that are needed for the desired functionality. In both cases, the overloaded functionality has to redirect the calls to the original library function after its completion. Figure 2 illustrates this setup.

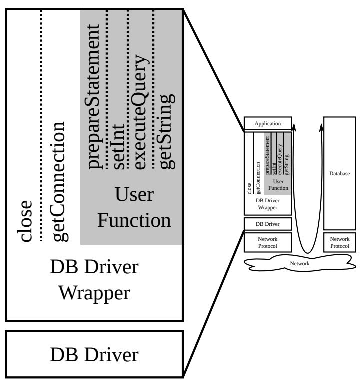

Figure 2: Partly changed driver to integrate user functionality.

It opens up the interaction of the application with the database to user manipulations. To give an idea of what is possible, we use the OCI API (Melnick, 2009) function `OCIPrepare`. This function gets the statement that is to be sent to the database. If we overload the function to manipulate the statement we can make the application do different things. We could introduce a new selection after the ones already present, we could change/amend the `WHERE` conditions or redirect the database tables from which the statements wants to take the data. The associate API functions in our abstraction layer would be `on_prepare_request` and `on_prepare_reply`.

A very basic example of the possibilities, is the amendment of the projection elements in a statement. We additionally want to get the rowid (physical address of a data record in Oracle):

```
SELECT id, name FROM employees
```

This statement could be changed to the following:

```
SELECT id, name, rowid FROM employees
```

As we also are able to modify the functions to fetch the data–using our API method `on_fetch_reply`–we can easily filter out the additional element to hide this change from the application.

3.2 Network-level Integration

All the actions, an application does on the API level, have to be transported to the database. For this, in most cases, a network connection is used. Database vendors develop different protocols to interact with their databases. Oracle introduced TNS, Microsoft TDS and IBM the DRDA for DB2.

We developed a network proxy, that is capable of understanding those protocols and gives an abstraction of the actions observed in the network. For this, the applications simply enter this proxy as their new database server and the proxy then relays the packets to the "real" database system.

Our proxy is capable of presenting the actions, that are observed in the network traffic, to user written modules. These actions are:

- preparation of statements
- binding of parameters
- execution of statements
- fetching results
- allocation and disposal of handles

As the network communication mirrors the actions on the API-level, it is possible to transform the API-calls to that abstraction. We have shown the feasability of this approach in (Phoonsarakun et al., 2013). Using this technique enables us to use the same modules on API as well as the network integration.

4 INTEGRATION EXAMPLES

Having shown how functionality can be introduced in an application-database-environment, we also want to give an example, how to use this technique to integrate real functionality. As mentioned in the introduction, we want to show this with an index and a statement cache.

4.1 Index Integration

At first, we have a closer look at how an index accelerates queries inside a database. An index is a data structure containing locations of data records ordered by attributes. We e. g. could have employees with names, identifiers and certain salaries. The table is ordered according to the employees' identifiers. If a query only incorporates the salaries inside its predicates, all data records have to be visited to determine if the predicate is true. Having an index on the salaries, the query would first go to that index which in turn delivers the data records locations that fulfill the predicate. The index lookup is much faster due to its ordering according to the attribute it is searched for. It returns the relevant data records. Reading those is also faster than reading all records, if the index lookup only turns up a fraction of all data. Inside databases this is used widely. The next sections describe the different aspects of having a database-external index.

4.1.1 Index Lookups

Our index, the ICIx, does not reside inside the database. However, we can use the same techniques that the database system uses inside to accelerate statements. Have a look at the following statement on our employee example from above:

```
SELECT name, id
FROM   employees
WHERE  salary > 2000
```

Let us further assume, that the database does not have an index on the salary column. We could amend the statement with the identifiers of the employees to give in a predicate, that is indexed by the database in the following way:

```
SELECT name, id
FROM   employees
```

```
WHERE  salary > 2000
   AND id IN ( 17, 19, 29)
```

The database systems optimizer identifies a predicate on a column that is indexed and will prefer it. After the selection on the id the remaining predicate is only to be evaluated on a very reduced data set and thus, fast.

We observed that the database system's accepted length of statements is limited and that the time for simply parsing the statement increases at least linearly with the length of the statement. To integrate the index results, we implemented a slight modification of the above approach. We insert our results into a separate table and replace the IN list with a subselect. The final statement for the above example then looks like this:

```
SELECT name, id
FROM   employees
WHERE  salary > 2000
   AND id IN ( SELECT idx_result FROM tmp_tbl)
```

Figure 3 shows the way the index is integrated into the communication of the application with a database. The kind of integration–library or proxy–does not matter, as only the abstract functions introduced earlier are used.

4.1.2 Index Maintenance

Up to now, we just had a look at how an index may deliver results to amend a statement. Most real data sets change over time. Database internal indexes are created, deleted and updated as the corresponding table changes. Further consistency conditions have to be met.

We found the Extensible Indexing Frameworks of databases (Stolze and Steinbach, 2003; Belden et al., 2009) as a good point to be integrated. These frameworks offer callback functions, that are called from the database system. This is done in a way that the ACID conditions are met. We use these callbacks, that are called in the events of creation, deletion, update and transactional operations to manage our index outside the database.

4.2 Statement Cache

Using our framework, it becomes possible to cache the results of statements, that are repeatedly executed. Allthough databases already implement such functionality, dependent on the application, we found, that some statements are executed over and over, without ever to expect any different result. Every statement triggers a database roundtrip, even with a database-side result cache. So what we did was to remove that

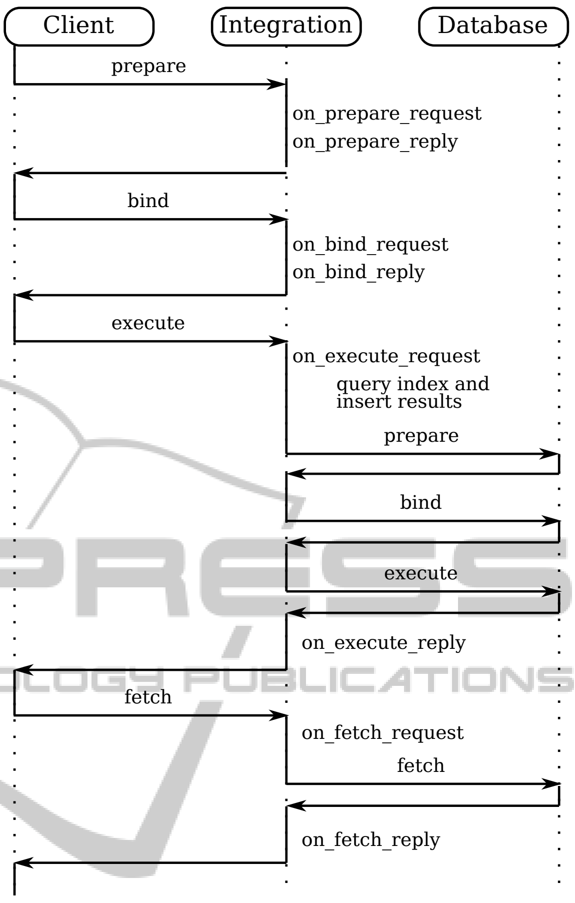

Figure 3: Scheme of how an index can be integrated into the communication of an application with a database.

roundtrip. In local networks that saves around 1 ms of network time, in wide area networks this time increases, depending on the network quality and technology.

Our approach is to intercept the function calls, that set up the statement execution. In our API these are `on_prepare_...`, `on_bind_...` and `on_execute_...`. We cache only statements, that are listed in a configuration, so to not accidentally alter statement results. If the statement matches a configured one, then we either execute it, in case the statements parameters were not already cached, and store the results. Otherwise, we answer the query from our cache.

5 RESULTS

For the results we want to concentrate on our framework. There are two major parts, the API integration and the network integration. All our tests were con-

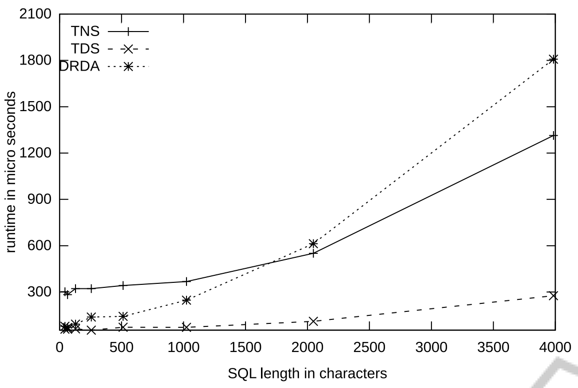

Figure 4: Times for different SQL lengths and protocol types.

ducted on an Intel Core i5 with 3 GHz and 4 GB of main memory running Debian 6 64 bit.

For the API integration, there is one relevant time, the overhead for the call into our framework and back to the "real" API. We measured function calls with approximately $3\mu s$. When an overloaded function is called within our framework, the only overhead is one additional call to the real function as well as the time needed to run the code inside.

The more interesting analysis concerns the network level integration. The proxy introduces one additional network hop into the communication of the application with the database server. In our local area network we measured this with approximately $100\mu s$ per message. As the proxy needs to analyze the traffic to and from the database server, this time is doubled for each communication step.

A second time, introduced by the proxy is the time needed to understand the data that was sent. Figure 4 shows the times needed to parse SQL statements of different length using different database protocols. It is clearly visible, that the various protocols need different time to process. This is due to the amount of work that has to be put into the parsing of the protocol packets. For TNS we had to reverse engineer the semantics and have some testing in the protocol analysis. This makes TNS the hardest to understand. DRDA then at increasing packet lengths shows increasing parsing times because of the inherent complexity. Of course the times also increase with the length of the statement. At the network layer this effect also increases the overall response times, as a packet has to first fully arrive at the proxy before it can be parsed and forwarded to the database system.

For the test with the index, we used our employees example. A table with an index only on the id column, and a statement, whose `WHERE` condition uses an unindexed attribute `salary`. We could show that the index worked the way we showed above. We only give a short overview of the results as they vary depending on the database system, its configuration and the hardware environment.

The statement cache worked also as expected. We could almost completely eliminate the network roundtrip times. The cache we implemented did a simple string matching of the statements and was very fast. For a number of less than 1000 statements in the cache, the lookup times were regularly less than $1ms$.

6 FUTURE WORK

Our future work has two directions, a technological and a systematic one. For the technological part, we want to extend our framework to support more database systems. The market is in constant change and so will be this technological foundation. Then there is the speed of the integration. Our results show delays introduced with our technology, especially using the network proxy. We want to minimize these times.

For the systematic part, we have to futher abstract the low level APIs of database vendors, especially when it comes to specialized functions such as the attributes of queries and parameters. Currently we ignore these and are thus limited in the application of our framework.

We also want to explore more ways to integrate results of our index. This is not always transparent to the application. We could e. g. organize the table in the database in a way that corresponds to the internal structure of our index, this could drastically lower the read times inside the database. In this scenario also automation is of interest for us. We need to incorporate metrics to decide whether or not to use our index with certain queries.

REFERENCES

(1995). International Standard for Database Language SQL - Part 3: Call Level Interface.

(2009). *PostgreSQL 8.4.3 Documentation*. The PostgresSQL Global Development Group.

(2013). http://www.greensql.com.

(2013). *Tabular Data Stream Protocol*. Microsoft Corporation.

Aboelfotoh, M., Dean, T., and Mayor, R. (2009). An empirical evaluation of a language-based security testing technique. In *Proceedings of the 2009 Conference of the Center for Advanced Studies on Collaborative Research*, pages 112–121. ACM.

Belden, E., Chorma, T., Das, D., Hu, Y., Kotsovolos, S., Lee, G., Leyderman, R., Mavris, S., Moore, V., Morsi, M., Murray, C., Raphaely, D., Slattery, H., Sundara, S., and Yoaz, A. (2009). *Oracle Database Data Cartridge Developers Guide, 11g Release 2 (11.2)*. Oracle.

Blakeley, J. (1997). In *Compcon '97. Proceedings, IEEE, title=Universal data access with OLE DB*, pages 2–7.

Görlitz, O. (2005). *Inhaltsorientierte Indexierung auf Basis künstlicher neuronaler Netze*. PhD thesis.

Guo, L. and Wu, H. (2009). Design and implementation of TDS protocol analyzer. In *Computer Science and Information Technology, 2009. ICCSIT 2009. 2nd IEEE International Conference on*, pages 633–636.

IBM (2009). *SQL Reference, Volume 1*. IBM Corporation.

Kansy, T. and Schwichtenberg, H. (2012). *Datenbankprogrammierung mit .NET 4.5: Mit Visual Studio 2012 und SQL Server 2012*. .NET-Bibliothek. Hanser Fachbuchverlag.

Leuoth, S., Adam, A., and Benn, W. (2010). Profit of extending standard relational databases with the Intelligent Cluster Index (ICIx). In *ICARCV*, pages 1198–1205. IEEE.

Melnick, J. (2009). *Oracle Call Interface Programmer's Guide, 11g Release 2 (11.2)*. Oracle.

Menon, R. (2005). *Expert Oracle JDBC Programming*. Apress.

Phoonsarakun, P., Adam, A., Lamtanyakul, K., and Benn, W. (2013). Extensible Database Communication Modification Framework. In Singh, R. K., editor, *Proceedings of the Second International Conference on Advances in Information Technology AIT 2013*.

Rich, B. (2011). *Oracle Database Reference, 11g Release 2 (11.2)*.

Stolze, K. and Steinbach, T. (2003). DB2 Index Extensions by example and in detail, IBM Developer works DB2 library.

The Open Group (2011). *DRDA V5 Vol. 1: Distributed Relational Database Architecture*.

SCITEPRESS
SCIENCE AND TECHNOLOGY PUBLICATIONS