© 2016 by the authors; licensee RonPub, Lübeck, Germany. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution license (http://creativecommons.org/licenses/by/4.0/).

Research online Publishing
www.ronpub.com

Open Access

Open Journal of Databases (OJDB)
Volume 3, Issue 1, 2016
www.ronpub.com/ojdb
ISSN 2199-3459

XML-based Execution Plan Format (XEP)

Christoph Koch

Computer Science Institute, Friedrich-Schiller-University Jena, D-07743 Jena, Germany,
christoph.koch@uni-jena.de

ABSTRACT

Execution plan analysis is one of the most common SQL tuning tasks performed by relational database administrators and developers. Currently each database management system (DBMS) provides its own execution plan format, which supports system-specific details for execution plans and contains inherent plan operators. This makes SQL tuning a challenging issue. Firstly, administrators and developers often work with more than one DBMS and thus have to rethink among different plan formats. In addition, the analysis tools of execution plans only support single DBMSs, or they have to implement separate logic to handle each specific plan format of different DBMSs. To address these problems, this paper proposes an XML-based Execution Plan format (XEP), aiming to standardize the representation of execution plans of relational DBMSs. Two approaches are developed for transforming DBMS-specific execution plans into XEP format. They have been successfully evaluated for IBM DB2, Oracle Database and Microsoft SQL.

TYPE OF PAPER AND KEYWORDS

Regular Research Paper: *Relational DBMS, execution plan, operator, standard, XML, format, SQL*

1 INTRODUCTION

Within the world of relational databases SQL is the standard descriptive query language, supported by almost every relational database management system (DBMS). Because the language standardizes only the logical DBMS layer, physical details as well as system internals are beyond the scope of SQL. Nevertheless even these non-standardized areas are relatively similar for common DBMSs. This also takes effect for the way cost-based optimization works in such systems: Multiple execution plans are built and, based on calculated expected costs, the potentially cheapest plan is executed for query processing. This “cheapest plan” in general could be externalized to simplify and visualize SQL tuning.

However, despite all the previously mentioned similarities, the output format of the execution plan is quite different for different DBMS. The plan details as well as the contained plan operators vary across the systems. To give an impression of these differences, Figure 1 shows two visual execution plans for the same SQL statement – Statement 3 (see Figure 2) of the TPC-H benchmark [21]. These two execution plans are built by IBM DB2 and Microsoft SQL Server respectively and visualized by their administration tools. Such execution plans cannot be directly exchanged and shared among different DBMSs, and the database administrators and SQL developers will also be burdened by the big differences.

In previous work [15] we showed that contrary to the differences in format, the main content of the execution plans of most common DBMSs is very similar. Main content refers plan and plan operator details which are not closely coupled to DBMS specifics. For example, almost every DBMS execution plan contains assumptions for CPU, I/O or overall costs

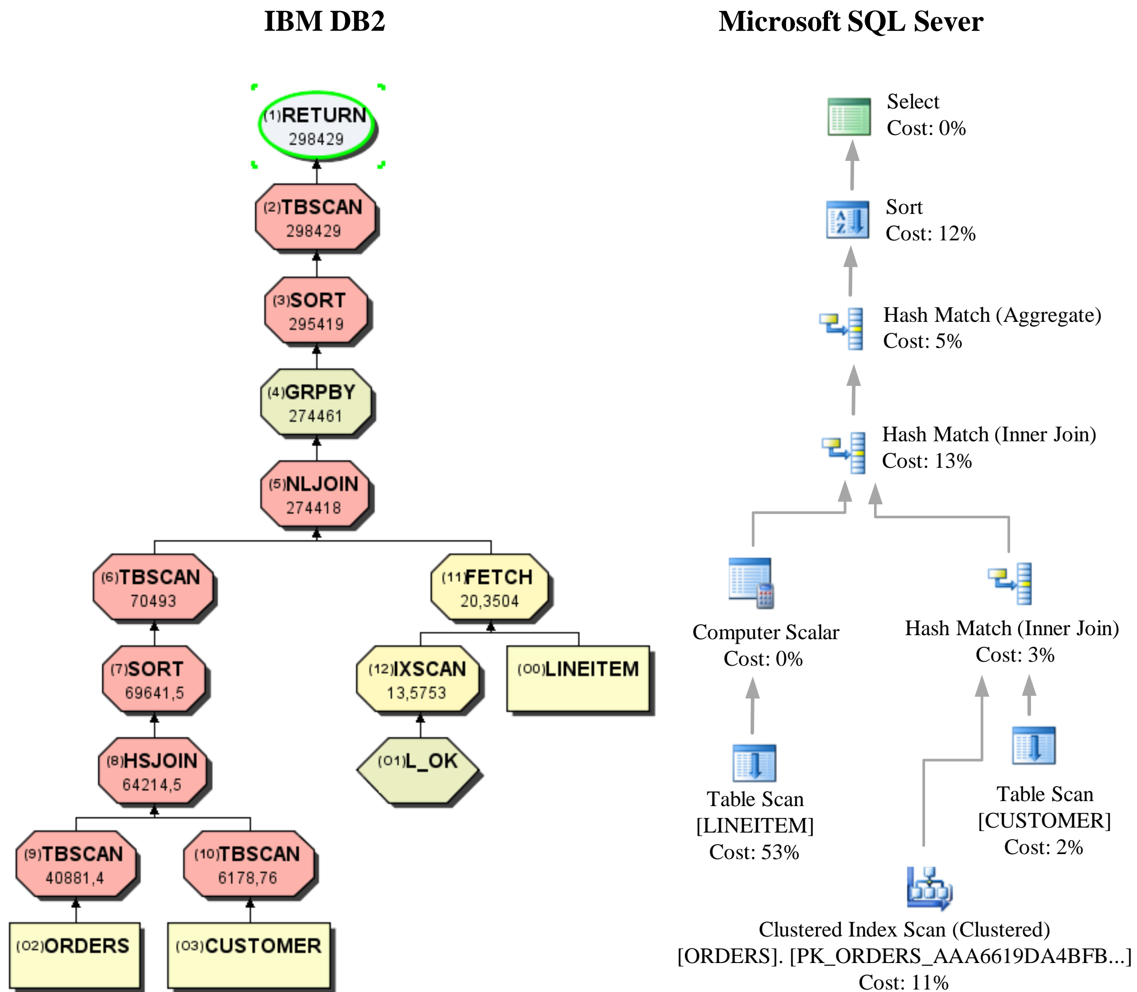


Figure 1: Two execution plans from different DBMSs for the SQL statement in Figure 2

as well as common details like the number of rows, projection lists, cost information or aliases for each execution plan operator. Furthermore, [15] also showed that for the used DBMSs – on an abstract level – their specific execution plan operators like a table or index scan are very similar. Because they are currently presented in different proprietary formats, there is an open space for creating a standard execution plan format. It might not be suitable to create a full format, which covers all DBMS specifics, and thus we want to create a light-weight standard execution plan format, which will contain general execution plan information. In our current paper, we want to build such a light-weight format based on the Extensible Markup Language (XML) [22], which provides important benefits of exchangeability and readability. Therefore, we name our format as XML-based Execution Plan format (XEP).

XEP can have multiple applications. One is to simplify basic SQL-tuning[1] for database administrators and SQL developers working with multiple DBMSs. If XEP is supported by the systems, no ongoing rethinking among specific execution plan formats will be necessary. The simplified characteristic of XEP also makes it easier for non-tuning experts like application developers to understand SQL execution plans. If a graphical XEP representation layer is developed in future, this benefit will further increase. In this context it would be possible with XEP to build DBMS-independent tools for execution plan analysis. Such tools could better support application developers and

[1] We want to address SQL tuning where DBMS specifics are less important; e.g. to notice materializations in access plans which in general are bad for SQL performance.

```
SELECT L_ORDERKEY,
       SUM(L_EXTENDEDPRICE*(1-L_DISCOUNT)) AS REVENUE,
       O_ORDERDATE, O_SHIPPRIORITY
FROM CUSTOMER, ORDERS, LINEITEM
WHERE C_MKTSEGMENT = 'BUILDING' AND
      C_CUSTKEY = O_CUSTKEY AND
      L_ORDERKEY = O_ORDERKEY AND
      O_ORDERDATE < '1995-03-15' AND
      L_SHIPDATE > '1995-03-15'
GROUP BY L_ORDERKEY, O_ORDERDATE, O_SHIPPRIORITY
ORDER BY REVENUE DESC, O_ORDERDATE
```

Figure 2: Statement 3 of TPC-H benchmark [21]

database administrators during SQL tuning than existing ones which currently support only proprietary DBMSs.

For example a tool, which will be built on top of XEP, could DBMS-independently analyze a set of execution plans and automatically determine most inefficient plans and plan operators within these plans. In this way, application developers and database administrators do not have to analyze all plans on their own. Instead, they can focus on the determined potential performance critical plans and plan operators and how to tune them properly.

Another important capability of XEP is that it can significantly improve the corporation between federated DBMSs because XEP is tended to be understandable and exchangeable by different DBMSs. [15] showed that execution plans across different DBMSs are reduced to rudimentary single remote operators to represent the whole remote processing part as a kind of black box. With the use of XEP instead of such primitive remote operators, whole remote operator "chains" could be communicated among XEP-supportive DBMSs.

In contrast to these potential applications, there is one thing for which XEP currently does not intend. XEP aims to standardize the representation of execution plans of different DBMSs, not their cost models. Therefore, execution plan comparisons between different DBMSs are not possible so far. This means that for example an execution plan of Oracle with costs of 10 is not automatically more efficient than an execution plan of DB2 with costs of 12.

In order to better understand the purpose of XEP, we also want to give a short overview about how query processing in DBMSs takes place and at which point XEP applies. Essentially, there are four different steps of query processing (see Figure 3) [9]. In the first step of query processing, a DBMS performs basic checks on a SQL query, including verifying the syntax of SQL statements, and translating the query into semantically equivalent relational algebra expression (i.e. a logical query plan) for efficient query optimization. The query optimization decomposes in two steps: The logical and the physical optimization.

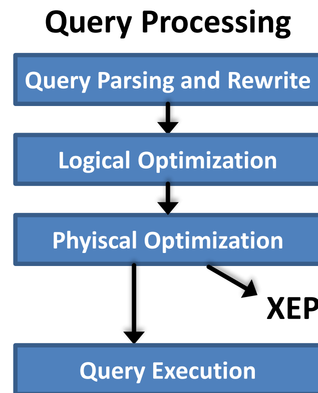


Figure 3: Steps of query processing

The logical optimization attempts to build a best relational algebra tree (i.e. a local execution plan) for the query. The best execution plan is defined as the plan with the lowest cost among all considered candidate plans. Based on the best logical execution plan, the physical optimization generate a DBMS-specific execution plan. In the last step of query processing, this DBMS-specific execution plan is executed to compute the results of the query. XEP aims to standardize the representation of the DBMS-specific execution plans. Therefore, XEP becomes relevant directly after physical query optimization and does not affect all processing steps before.

The rest of the paper is structured as follow: Section 2 describes in detail the content of XEP. Section 3 focuses on the XEP-underlying XML schema 1.1 [23, 24] document, its specifics and its structure. Section 4 addresses the implementation details how to transform DBMS-specific execution plans to XEP format. Section 5 discusses related work. Section 6 summaries and concludes this work.

2 XEP Content

The idea behind XEP was to design a light-weight format, which standardizes the (representation of) most important and common execution plan details, is easily exchangeable among multiple systems and also easily readable by these systems as well as by humans. Therefore, it uses XML technology and does not contain DBMS-specific information. Instead, XEP handles execution plans and plan operators on an abstract level. In XEP, general plan information is captured within an XEP *executionPlan* object and different plan operators are captured by corresponding XEP operator objects. A overview of XEP content is outlined in Figure 4. This figure and the following figures about XEP operators are represented using the notation of Unified Modeling Language (UML) [18].

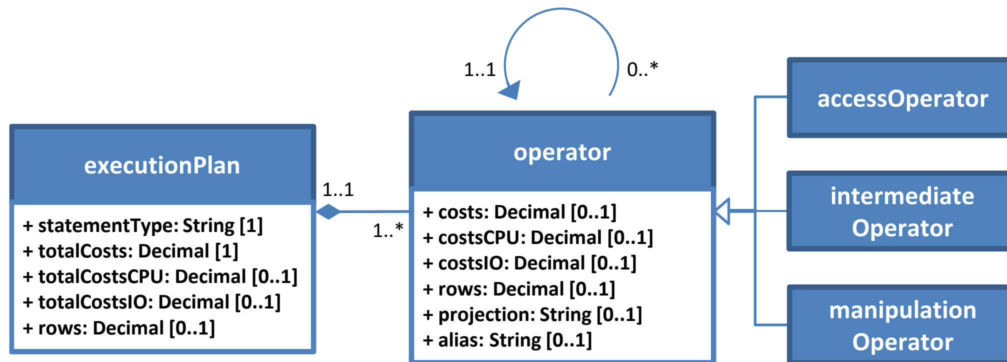


Figure 4: XEP overview

Each XEP execution plan consists of exactly one executionPlan object (see section 2.1). This object contains one overall parent XEP operator. This operator can contain arbitrary other XEP operators, which can also contain arbitrary other operators and so on. All XEP operators are classified into three categories: *accessOperator* that consists of the operators for data access, *intermediateOperator* that comprises the operators for processing intermediate results, and *manipulationOperator* for data manipulation. The three categories of operators will be detailed in the following subsections.

2.1 General Details of Execution Plan

XEP includes one central *executionPlan* object, which contains general details for the execution plan of an SQL statement. The object is intended to represent plans for SQL data query language (DQL) and data manipulation language (DML), so it supports SELECT, INSERT, UPDATE, DELETE and MERGE statement types. The statement type (*statementType*), treated as string as shown in Figure 4, is one of the *executionPlan* object attributes. All other attributes are related to total costs of different items, which are calculated by the optimizer for the analyzed SQL statement. These cost attributes are handled as decimal value, and reveal the expected total efforts to process the whole statement. The attribute *rows* indicates an expected number of rows returned from execution of the statement.

Although it is obvious that there cannot be fractions of a row, the number of rows in most DBMS execution plans is handled as a decimal value, so XEP simply conforms to the convention. Regarding cost information XEP distinguishes between the overall (*totalCosts*), CPU (*totalCostsCPU*) and I/O (*totalCostsIO*) costs. [15] has showed that CPU and I/O costs are not provided by all of the considered DBMSs, so the corresponding attributes of the *executionPlan* object for the cost information are marked as optional. For the same reason the attribute *rows* for the number of rows is also optional.

2.2 Operators of Execution Plan

Modern DBMSs support a number of different operators of execution plan. [15] classified these operators as data access operators (*accessOperator)*, the operators for processing intermediate results (*intermediateOperator)* and data manipulation operators (*manipulationOperator)*. With a few modifications, which we will explain in the next subsections, XEP also uses this classification and the operators in it. Independent of a specific operator there is common operator content in XEP. Analogous to the *executionPlan* object, the common content contains several attributes for describing different cost information (*costs*, *costsCPU*, *costsIO*), and the number of rows (*rows*) that the operator is expected to return after processing.

However, different from the *executionPlan* object, the cost attributes of an XEP operator represent the costs for processing only the operator. Therefore, they are not cumulated cost information items. A XEP operator also owns an attribute of *projection* list and an attribute of *alias*. The *projection* list consists of all columns that are returned by the operator. The *alias* attribute contains the identifier of an object or a subquery reference to distinguish among parts of execution plan in cases where, for example, same objects are involved several times but with different aliases.

All attributes of XEP operators are optional. This is due to several reasons. First, [15] pointed out that for almost every attribute there is one DBMS that does not provide it. Secondly, even if a DBMS takes an attribute into consideration, it is not unusual that its information is missing in special situations. To become a standard format, XEP intends to support all possible scenarios and consequently treats all attributes as optional ones.

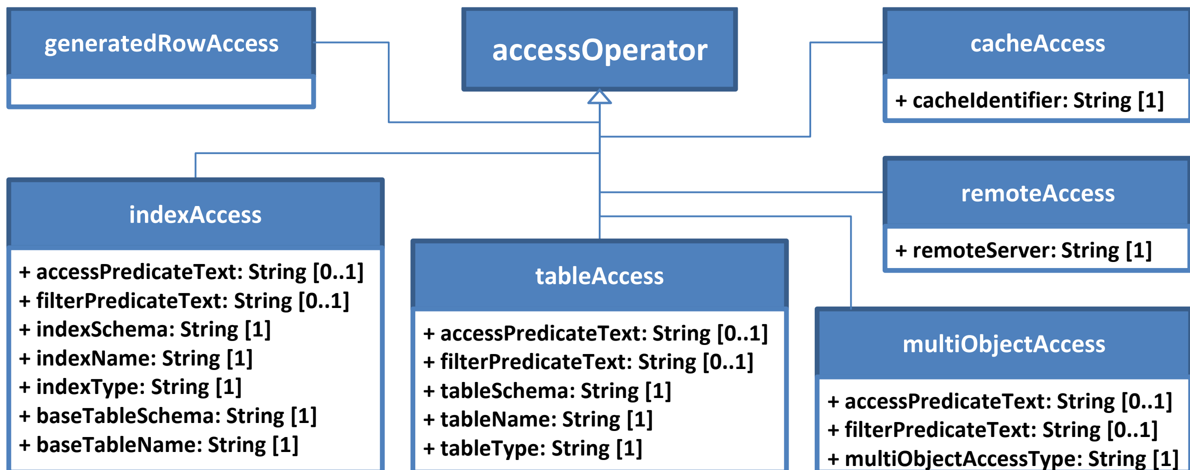


Figure 5: XEP operators for data access (*accessOperator*)

As shown before in Figure 4, XEP operators can contain other operators as well. Different from the proprietary standards like Microsoft SQL Server SHOW PLAN XML format [17], which in simplified terms only describes the presence of operators, XEP also defines the relationship between different operators. This means that for each operator XEP describes valid child operators and the number of possible operator children. For example, XEP allows an *indexAccess* operator to appear as child of a *tableAccess* operator. That is because in many cases, where an index is used in the next step, additional column data needs to be accessed from its base table by using the row identifiers read from this index. Furthermore, XEP also requires *intermediateOperators* to have at least one child operator, and XEP comprises many such dependencies. Further detailed information for XEP operators is described in our XEP-Schema document [14], which is online free available.

The subsequent sections will give detailed explanation for all XEP operators expect for one special operator *otherOperator*. As mentioned several times before, XEP does not intend to be and cannot be an overall standard format for all DBMS-specific execution plan details. Therefore, it only standardizes common similar plan operators, which are the vast majority of operators, but there are a few operators that XEP does not support, e.g. OLAP operators like Cube Scan, Pivot or Unpivot (Oracle), parallel processing operators like Partition (Oracle), Parallelism (SQL Server), Partition or Repartition (DB2).

To maintain a proper relationship among XEP operators within an XEP execution plan, these unsupported operators need to be included. XEP therefore handles them as one generic operator *otherOperator*, which is allowed at almost every position within an execution plan, and it owns only the attributes of the generalized XEP operator as shown in Figure 4.

2.2.1 Data Access Operators (*accessOperator)*

The XEP *accessOperator* category contains the operators for data access, which are outlined in Figure 5. Different kinds of data is accessed by different access operators. The rows in a table is accessed by *tableAccess* operators, the entries of an index by *indexAccess*, the generated rows in memory by *generatedRowAccess*, the cached contents by *cacheAccess*, and the data on a remote server by *remoteAccess*. Additionally XEP also supports the simultaneous access of several data objects by the multiObjectAccess operator. Compared to [15] *cacheAccess* and multiObjectAccess are new access operators added to the category for XEP.

Except for *generatedRowAccess* all other access operators contain additional attributes. For *tableAccess* the accessed table is listed with its schema, name and type. The type differentiates among a standard table, a temporarily created table, a materialized query table (in some DBMSs also known as materialized view or indexed view), a table function result, a transition table and an external table that for example could be built on external csv files [10].

Besides *accessPredicateText* attribute, *tableAccess* operator also contains one *filiterPredicateText* attribute, which is used to filter parts of data. *accessPredicateText* is directly applied while accessing some data, and *filterPredicatetext* is applied right after the data is accessed. Because there could be more than one *accessPredicateText* or one *filterPredicateText*, the predicates from each type are put in conjunction and then handled as a whole conjunction String.

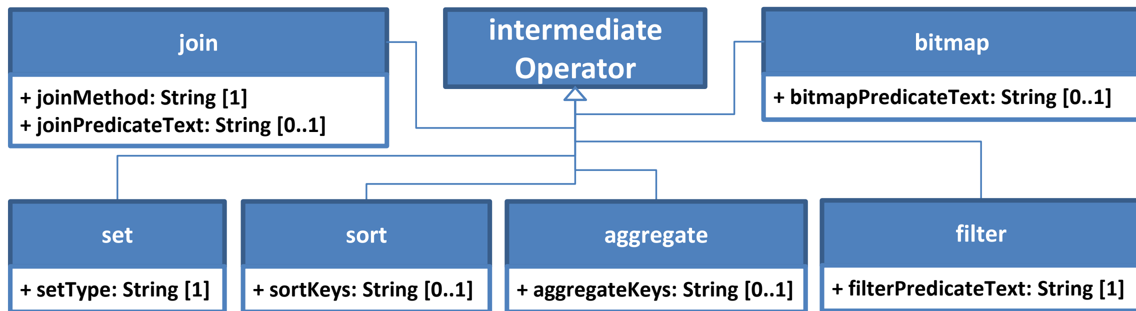


Figure 6: XEP operators for processing intermediate results (*intermediateOpertor*)

The *indexAccess* and the *multiObjectAccess* operator also own these two types of optional predicate attributes. Like *tableAccess, indexAccess* also contains additional attributes that identify the accessed index by means of schema, name and base table schema and base table name. An *indexAccess* object also contains information about the type of index. XEP for the sake of simplicity differentiates only among standard index, bitmap index, index organized table, temporarily created index and bloom filter (that strictly speaking is not a real index).

In addition to the two optional predicate-related attributes, the *multiObjectAccess* object also includes a mandatory attribute *multiObjectAccessType*, which describes the type of a *multiObjectAccess* object. This type can be *rowSet* and *rowIdSet*. The *rowSet* type indicates the simultaneous access to whole rows of multiple objects, and *rowIdSet* means only access to identifiers of rows from the objects

The *remoteAccess* operator only contains one additional attribute *remoteServer* to identify the server at which the remote data is located. The *cacheAccess* operator also only owns one additional attribute *cacheIdentifier* that identifies appropriate cached results.

2.2.2 XEP Intermediate Operators (*intermediateOperator*)

The *intermediateOperator* category contains the operators for further processing of data accessed before. For such processing, XEP supports these intermediate operators: *join*, *bitmap*, *set*, *sort*, *aggregate* and *filter* as shown in Figure 6.

A *join* operator is used to join two (one left/outer and one right/inner) interim results. It features a join method and a join predicate text. XEP supports the following join methods: nested loop, merge, hash and bitmap join. For the sake of simplicity, XEP also treats bloom filter usage as a kind of join between table data and a bloom filter. All other mostly DBMS-specific join methods are captured in XEP by a generic method called *otherJoin*. The *jointPredicateText* attribute of a join operator is similar to the *accessPredicateText* or filterPredicateText attributes in the access operators in the previous subsection. Therefore, if there are multiple join predicates, they will be put into conjunction and handled as one string. This is also true for the *bitmapPredicateText* attribute of the *bitmap* operator and *filterPredicateText* of *filter* operator

A *bitmap* operator implements bitmap processing, i.e. interim results are processed depending on some earlier created bitmap or some previously accessed bitmap index data. The way bitmaps are used by the *bitmap* operator (bitmap AND, bitmap OR and others as well as arbitrary combinations of them) is described by the *bitmapPredicateText* attribute, which therefore contains a logical expression.

At best filtering takes place directly at data access as described in section 2.2.1. Furthermore, interim results can also be filtered after data access by using a *filter* operator. The way of filtering data has to be defined as a filter predicate, which therefore is a mandatory attribute for XEP filter operator.

To combine multiple interim results as union, intersection or exception, XEP provides a *set* operator. The intended type of set operation is represented by a mandatory attribute called *setType.*

Other two XEP intermediate operators are *sort* and *aggregate*. The *sort* operator processes different types of interim results and the *aggregate* operator is responsible for data aggregations. Both operators have similar structures. The *sort* operator has a *sortKey* attribute that provides information of keys being sorted for and the *aggregate* operator requires an *aggregateKey* attribute. The aggregate key is used analogously to the columns that the aggregation should process. *sortKey* and *aggregateKey*, are both optional attributes. If a key is missing, then the current interim result is sorted or grouped using all available columns.

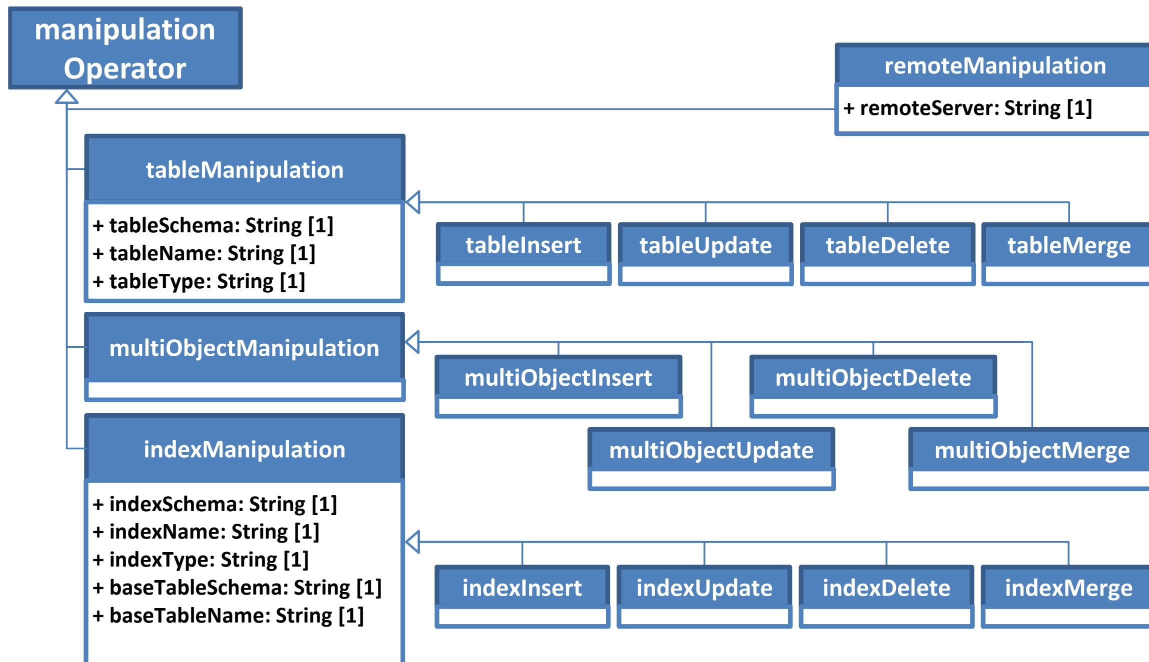


Figure 7: XEP operators for data manipulation (*manipulationOperator*)

2.2.3 XEP Manipulation Operators (*manipulationOperator*)

The operator category *manipulationOperator* consists of operators, which are directly responsible for data manipulations in database objects like tables or indexes. Like execution plans of common DBMSs, XEP supports table and index manipulation by INSERT, UPDATE, DELETE and MERGE operators. XEP also defines four multi-object manipulation operators and a special remote manipulation operator. Figure 7 shows all manipulation operators in the operator category.

A *tableInsert* operator is used to insert one or more data rows in a table, which is identified by its schema, name and type. All these attributes are treated as mandatory string values. Analogous to the *tableInsert* operator, the *tableUpdate* operator processes update of one or more data rows, the *tableDelete* operator deletes one or more data rows and the *tableMerge* operator merges one or more data rows. All these operators contain the same table attributes as the *tableOperator* operator.

The processing of index is handled in a similar manner as the processing of table. The *indexInsert* operator describes the insert operation one or more data rows in an index, *indexUpdate* operator performs the update of one or more data rows of an index, the *indexDelete* operator deletes one or more data rows of an index, and the *indexMerge* operator merges one or more data rows of an index. Each of these index manipulation operators contains several attributes to identify the manipulated index and its base table. These attributes are listed in the *indexManipulation* operator, and they are mandatory and represented as string.

As one of the four multi-object manipulation operator, *multiObjectInsert* is used to express the simultaneous insertion of one or more data rows in several tables or indexes. Each considered table/index, where an insertion takes place, is treated as a separate *tableInsert*/*indexInsert* child operator. Therefore, the *multiObjectInsert* operator does not need additional attributes. For update, deletion and merge processing, XEP provides the following operators: *multiObjectUpdate*, *multiObjectDelete* and *multiObjectMerge*. These attributes are analogous to the operators for the processing of tables, and also work in a similar manner.

Similarily to the *remoteAccess* operator described in section 2.2.1, XEP additionally provides a *remoteManipulation* operator. The operator represents manipulations, which are processed on a remote server. The server itself and its location are identified by the attribute *remoteServer*.

3 XEP SCHEMA

One of the main goals of designing XEP is to provide a format for execution plans, and make them easily exchangeable among multiple systems and easily readable by these systems as well as by humans. Exchangeability and readability are two of the biggest advantages of XML, so we adopt this technology for XEP. To describe the elements in an XML document and the structure among them, several techniques exist. Besides more rarely used technique like RELAX NG [6] or Schematron [11], the most used technique is XML schema [23]. For describing the structure of XEP, we also use XML schema and design a XEP-XML-Schema document, which is freely downloadable [14]. We will explain the schema in this section.

As mentioned in Section 2, XEP describes not only valid operators but also valid relationships between them. In order to put this into practice and to make it as modular and legible as possible, the salami slice [7] XML schema design for structuring the XEP schema is used. This means that each element (in our case mostly an operator) is declared as a separate complex type component as e.g. shown in the sort operator declaration in Figure 8. To achieve relationship definitions in the sense of valid child nodes, these components are assigned to identically named elements, which are assembled in sequence or choice XML schema elements.

XML allows putting information into elements, attributes and unstructured text nodes. XEP uses elements of complex type to represent operators and attributes of simple types to store detailed operator information. Values for the attributes like *tableType*, *setType* and *joinMethod* are pre-defined ones. XEP does not allow data in text nodes. To combine similar operators to one group, XEP uses XML *substitutionGroup* in the XML schema document [23, 24]. The XEP schema defines one group for *accessOperator* when the value of the attribute *substitutionGroup* is "access", one for *intermediateOperators* ("intermediate") and also one for *manipulationOperators* ("manipulation"). Figure 9 shows some substitutionGroup assignments for selected operators.

A big advantage of this grouping is the simplification in defining "general" relationships between XEP operators. *SubstitutionGroup* together with XML schema *ref* constructs allows a whole group as a child node for an operator, and thus the definitions of separating children via an XML schema *choice* element are not necessary. Figure 8 also illustrates this behavior by the example of the complex type declaration of the sort operator.

```
<xsd:complexType name="sort">
  <xsd:complexContent>
    <xsd:extension base="_operator">
      <xsd:choice>
        <xsd:element ref="intermediate" />
        <xsd:element ref="access" />
        <xsd:element name="otherOperator“
                     type="otherOperator" />
      </xsd:choice>
      <xsd:attribute name="sortKeys“
                     type="xsd:string“
                     use="optional" />
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```

Figure 8: Complex type declaration of *sort* operator

```
<xsd:element name="tableAccess" type="tableAccess"
             substitutionGroup="access"/>
<xsd:element name="indexAccess" type="indexAccess"
             substitutionGroup="access"/>
...
<xsd:element name="join" type="join"
             substitutionGroup="intermediate"/>
<xsd:element name="set" type="set"
             substitutionGroup="intermediate"/>
...
<xsd:element name="tableInsert" type="tableInsert"
             substitutionGroup="manipulation"/>
<xsd:element name="indexInsert" type="indexInsert"
             substitutionGroup="manipulation"/>
...
```

Figure 9: Declarations of *substitutionGroup*

```
<xsd:assert
  xpathDefaultNamespace="##defaultNamespace"
  test="every $i in
         //tableAccess[not(*) and
                       @tableType = 'tempTable']
           /concat(@tableSchema,'.',@tableName)
        satisfies
         //tableInsert
           /concat(@tableSchema,'.',@tableName) = $i"
/>
```

Figure 10: *Assert* to guarantee operator dependency

XEP schema is based on XML schema 1.1 recommendation, because XEP also uses its *assert* elements to define detailed dependencies between operators, which cannot be defined with XML schema 1.0 techniques in the same easy way. As an example, Figure 10 describes an *assert* element, which defines the following rule.

If a tableAccess operator accesses a temporarily table and does not have any child operator, then somewhere else in the execution plan there should be a tableInsert operator inserting rows into the same table as referenced within the tableAccess operator.

4 Implementation and Evaluation

XEP tends to be a light-weight standard execution plan format for every relational DBMS. So proprietary DBMSs (with closed source codes) are also included. To enable XEP execution plans for each considered DBMS, we have implemented a transformer of execution plans on top of the DBMS interfaces, which transforms DBMS-specific execution plans to our XEP format. We have designed two approaches of transformation: an transformation application and Extensible Stylesheet Language Transformation (XSLT) [25], as showed in Figure 11.

Among the most common relational DBMSs, half of the systems (Oracle Database, Microsoft SQL Server, PostgreSQL) are able to export SQL execution plans in proprietary XEP-like XML format [15]. For these XML execution plan supportive DBMSs, we develop an XSLT-based approach as shown in the lower path in Figure 11. This approach uses XSLT stylesheets, which can transform an XML document from a format to another.

With the XSLT-based approach, a DBMS-specific execution plan is first exported into its proprietary XML format. This XML document and a DBMS-depending XSLT stylesheet are used as input of an (in general external) XSLT processor, which accomplishes the XEP transformation and outputs an appropriate execution plan in XEP format. With this approach, the core components are the XSLT processor (in our implementation we use Saxon [20]), and the XSLT stylesheet that has to be developed for each DBMS. In some of our previous work, we have successfully built an XSLT stylesheet for Oracle Database [13] and one for Microsoft SQL Server [4]. These stylesheets are online freely available [14].

Apart from these XML execution plan supportive DBMSs, [15] showed some DBMSs, like MySQL, IBM DB2 LUW, IBM DB2 z/OS, do not support XML plan output. Because these systems are at least able to export execution plans in a relational table structure, we use the application-based approach (in our implementation, a Java application is developed) for XEP transformation as shown in the upper path in Figure 11.

With the application-based approach, a DBMS-specific execution plan is first exported into a relational table. The transformation application reads the details of the execution plan from the relational table, and transform them according to DBMS specific rules to appropriate XEP objects. Once all data of the execution plan is processed, all XEP objects are serialized to one XML document, which represents a valid XEP execution plan. To make the transformation as much platform independent as possible, we developed several

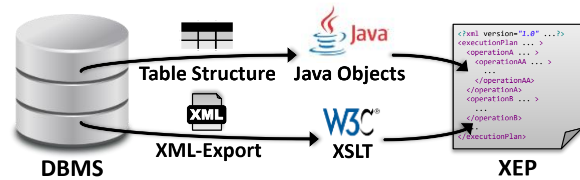


Figure 11: Approaches for XEP transformation

```
<?xml version="1.0" encoding="UTF-8"?>
<executionPlan xmlns="http://www.minet.uni-jena.de/
                      dbis/XEP" … >
  <sort … >
    <aggregate … >
      <join … >
        <left>
          <sort… >
            <join … >
              <left>
                <tableAccess name="ORDERS" … />
              </left>
              <right>
                <tableAccess name="CUSTOMER" … />
              </right>
            </join>
          </sort… >
        </left>
        <right>
          <tableAccess name="LINEITEM" … >
            <indexAccess name="L_OK" … />
          </tableAccess>
        </right>
      </join>
    </aggregate>
  </sort>
</executionPlan>
```

Figure 12: Shortened XEP example

DBMS-specific XEP mappers and the XEP serializer in Java using Saxon [20] library. Currently we have developed an XEP mapper for IBM DB2 LUW and one for IBM DB2 z/OS, and a general XEP serializer. The code of implementation is online freely available in [14].

The two mappers have been successfully tested with all queries of the TPC-H benchmark (Q1 – Q22) [21]. Figure 12 shows the XEP document (in shortened form nearly without attributes) for Statement 3 of the TPC-H benchmark (see Figure 2) [21], which was transformed from an IBM DB2 LUW execution plan. In addition to these tests, we also verified that the mapper for IBM DB2 z/OS was able to transform all DB2-specific execution plans for the 99 queries of the TPC-DS benchmark [21] into XEP format. We also tested this XEP mapper in the DB2 environment of DATEV eG, where it transformed all (dynamic) SQL statements (over 10,000 different statements) from DB2-specific execution plan format into XEP format successfully.

Independent of the different approaches after transforming DBMS-specific execution plans into XEP, all the resulting execution plans are successfully

validated based on the XML schema defined for XEP. The validator that we use is the Xerces XML schema 1.1 validator [2]. The validating result shows the correctness of two transformation approaches.

5 RELATED WORK

XEP is a format that aims to standardize the representation of query execution plans of relational DBMSs. In contrast, there are several techniques to standardize the access to these (and mostly other non-relational) DBMSs and therefore are consequently responsible for generation of execution plans. For example, these techniques include LINQ [16, 8] and [12] and scalaQuery [26]. XEP wants to create a standardized format for already generated plans to simplify various tuning activities. In these scenarios, it is not important by which (standardized) technique an execution plans was built.

As mentioned before in this paper, some DBMSs support XML representation of execution plans. These formats are mostly build on top of XML to the information of exection plans, e.g. the Microsoft SHOW PLAN XML format [17] and the XML format produced by the DBMS_XPLAN package in Oracle [19]. Therefore, using XML to describe execution plans of queries is not new. However, our work aims at creating a standard XML-format that is understandable by different DBMSs. There are also other formats for execution plans like JSON [5] and YAML [3] and other ones [15]. JSON, YAML as well as XML formats are supported by PostgreSQL [1]. All these formats are proprietary and only supported by the DBMSs where these formats are developed. XEP is the first XML-based format for representing execution plans from different DBMSs. Therefore, XEP is currently the only format that allows DBMS-independent execution plan analysis by humans as well as by external tools.

6 SUMMARY AND CONCLUSIONS

In this paper, we described XEP, a light-weight, easily exchangeable and easily readable standard format for SQL execution plans. To ensure its targets, XEP is built on XML technology, which provides the advantages of being easily exchangeable and readable. The content and structure of XEP is developed using XML schema language. Because of some specific concepts, XEP uses the XML schema 1.1 recommendation. Two approaches (application-based and XSLT-based) are developed for transforming DBMS-specific execution plans into XEP representation. The two approaches have been successfully evaluated on the DBMSs of *IBM DB2, Oracle Database and Microsoft SQL.*

There are several issues to work on in the future. Currently, XEP is only implemented for proprietary common relational DBMSs. Thus, implementations for open source systems like PostgreSQL or MySQL are missing today. Due to the public availability of their code bases, these systems offer even larger opportunities for XEP integration. Therefore, it should be possible to integrate the XEP execution plan format directly and deeper into the DBMS kernel, as it has been done for the proprietary XML, JSON or YAML format. Proprietary database vendors like Oracle, Microsoft or IBM could act in the same manner in the future.

If these steps are taken, then the investigations into federated access plans based on XEP should be intensified. Currently, XEP is built by external procedures, and such execution plan corporation would only be possible within the external layer. However, in terms of cross-DBMS optimization and similar issues this does not make much sense.

We want to highlight that XEP tends to be easier readable for humans than proprietary formats of execution plans. Because of its simplicity and its focus on important DBMS-independent information, the structure of XEP is very clear. However, it is not automatically a proof of its readability and understandability by human beings. These are some aspects – of course together with the predicted general added value of XEP and a useful graphical XEP representation layer – that should be investigated in future work.

Finally, we want to mention the content of XEP. Using the *asserts* property of XML Schema, a few detailed dependencies among XEP operators were described. However, there might be more dependencies, which should be determined and implemented to the XEP schema document. This also should be investigated in the future.

REFERENCES

[1] I. Ahmed, A. Fayyaz, A. Shahzad, “PostgreSQL Developer’s Guide”, Packt Publishing, February 2015.

[2] Apache Xerces, “The Apache Xerces™ Project”, http://xerces.apache.org/, accessed July 28, 2016.

[3] O. Ben-Kiki, C. Evans, I. döt Net, "YAML Ain't Markup Language (YAML™) Version 1.2", October 1, 2009. http://www.yaml.org /spec/1.2/spec.html.

[4] M. Birke. “Transformation of the SQL Server Showplan XML Format into the generic Execution Plan Format XEP” (in German),

Project Work, Faculty of Mathematics and Computer Sciences, Friedrich-Schiller-University of Jena, August 2015.

[5] T. Bray, "The application/json Media Type for JavaScript Object Notation (JSON)", March 2014. https://tools.ietf.org/html/rfc7159.

[6] J. Clark and M. Murata, "Relax NG specification", December 2001. http://relaxng.org /spec-20011203.html.

[7] R. L. Costello, “XML Schemas: Best Practices”, http://www.xfront.com/BestPractices Homepage.html, accessed July 28, 2016.

[8] S. W. Dietrich, “Is LINQ in your toolbox?”, In: ACM Inroads, 4(1):31-33, March 2013.

[9] G. Graefe, “Query Evaluation Techniques for Large Databases”, In: ACM Computing Surveys, 25(2):73-170, June 1993.

[10] Internet Engineering Task Force (IETF) Network Working Group, “Common Format and MIME Type for Comma-Separated Values (CSV) Files”, October 2005. https://tools.ietf.org/html/rfc4180.

[11] R. Jelliffe and Academia Sinica Computing Center, "The schematron assertion language 1.6", January 1, 2002. http://xml.ascc.net/resource/ schematron/Schematron2000.html.

[12] M. Karjalainen, G. J. L. Kemp, “Uniform Query Processing in a Federation of RDFS and Relational Resources”. In: Proc. *IDEAS'09*, September 2009.

[13] J. M. Keil. “Transformation of Oracle Execution Plans into the generic Execution Plan Format XEP” (in German), Project Work, Faculty of Mathematics and Computer Sciences, Friedrich-Schiller-University of Jena, October 2014.

[14] C. Koch. “XML-based Execution Plan format (XEP) resources”, May 2015. http://users.minet.uni-jena.de/~ma47get/XEP/xep.html.

[15] C. Koch, K. Büchse. “Execution Plans and Plan Operators of Relational DBMS” (in German), In: Proc. *GvDB'15*, May 2015. http://ceur-ws.org/Vol-1366/paper10.pdf.

[16] E. Meijer, B. Beckman, G. Bierman, “LINQ: Reconciling Object, Relations and XML in the .NET Framework”, In: Proc. *SIGMOD'06*, January 2006.

[17] Microsoft Corporation. “SQL Server 2014 – SET SHOWPLAN_XML (Transact-SQL)”, https://msdn.microsoft.com/de-de/library/ ms187757.aspx, accessed July 28, 2016.

[18] Object Management Group (OMG), “OMG Unified Modeling Language (OMG UML) Version 2.5”, OMG Specification, March 2015. http://www.omg.org/spec/UML/2.5/PDF

[19] Oracle, “Administrator’s Guide – 12c Release 1 (12.1)”, June 2016. https://docs.oracle.com/database/121/ADMIN/E41484-11.pdf.

[20] SAXON, “The XSLT and XQuery Processor”, October 2010. http://saxon.sourceforge.net/

[21] TPC. “TPC Benchmark H”, Standard Specification, November 2014. http://www.tpc.org/tpc_documents_current_versions/pdf/tpch2.17.1.pdf.

[22] World Wide Web Consortium (W3C), “Extensible Markup Language (XML) 1.0 (Fifth Edition)”, Recommendation, November 2008. http://www.w3.org/TR/xml/.

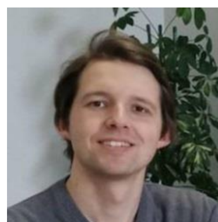
[23] World Wide Web Consortium (W3C), “XML Schema Definition Language (XSD) 1.1 Part 1: Structures”, Recommendation, April 2012. http://www.w3.org/TR/xmlschema11-1/.

[24] World Wide Web Consortium (W3C), “XML Schema Definition Language (XSD) 1.1 Part 2: Datatypes”, Recommendation, April 2012. http://www.w3.org/TR/xmlschema11-2/.

[25] World Wide Web Consortium (W3C), “XSL Transformations (XSLT) Version 2.0”, Recommendation, January 2007. http://www.w3.org/TR/xslt20/.

[26] S. Zeiger, “ScalaQuery”, http://scalaquery.org, accessed August 22, 2016.

AUTHOR BIOGRAPHIES



Dipl. Inf. Christoph Koch studied Computer Science at the University of Jena and earned his Diploma in 2012. Since 2012 he is an employee in the departure databases at DATEV eG in Nuremburg and also a researcher at the group of database and information systems at the university of Jena. His main research areas are proactive model-based database performance analysis and forecast.