DrillBeyond: Processing Multi-Result Open World SQL Queries

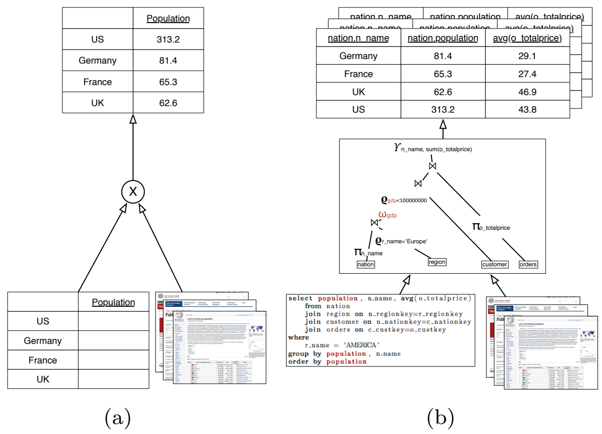
Julian Eberius, Maik Thiele, Katrin Braunschweig and Wolfgang Lehner
Technische Universität Dresden
Faculty of Computer Science, Database Technology Group
01062 Dresden, Germany
[firstname.lastname]@tu-dresden.de

ABSTRACT

In a traditional relational database management system, queries can only be defined over attributes defined in the schema, but are guaranteed to give single, definitive answer structured exactly as specified in the query. In contrast, an information retrieval system allows the user to pose queries without knowledge of a schema, but the result will be a top-k list of possible answers, with no guarantees about the structure or content of the retrieved documents.
In this paper, we present DrillBeyond, a novel IR/RDBMS hybrid system, in which the user seamlessly queries a relational database together with a large corpus of tables extracted from a web crawl. The system allows full SQL queries over the relational database, but additionally allows the user to use arbitrary additional attributes in the query that need not to be defined in the schema. The system then processes this semi-specified query by computing a top-k list of possible query evaluations, each based on different candidate web data sources, thus mixing properties of RDBMS and IR systems.
We design a novel plan operator that encapsulates a web data retrieval and matching system and allows direct integration of such systems into relational query processing. We then present methods for efficiently processing multiple variants of a query, by producing plans that are optimized for large invariant intermediate results that can be reused between multiple query evaluations. We demonstrate the viability of the operator and our optimization strategies by implementing them in PostgreSQL and evaluating on a standard benchmark by adding arbitrary attributes to its queries.

1. INTRODUCTION

The domains of information retrieval and database systems have been traditionally kept separate. The reasons for distinguishing two classes of systems are many: there are differences in the *type of data* that is managed, the *language* used to query the data as well as the nature of the *query result*, among other differences. Concerning the type data,



```
select population , n_name , avg(o_totalprice)
    from nation
    join region on n_regionkey=r_regionkey
    join customer on n_nationkey=c_nationkey
    join orders on c_custkey=o_custkey
where
    r_name = 'AMERICA'
group by population , n_name
order by population
```

Figure 1: Existing standalone augmentation queries (a) and the proposed augmentations operator integrated into relational query processing (b) returning ranked results

DBMS deal primarily with structured data, while the information retrieval systems deal primarily with unstructured data such as text. Database management systems work with fully specified queries in a structured language, while information retrieval systems accept queries given as a collection of keywords in the general case. Finally, DBMS return an exact single answer, while answers in information retrieval are usually uncertain and therefore a ranked list of possible answers is returned.
The typical usage scenarios also vary. A DBMS is more useful in analytic scenarios, but requires a deep understanding of the schema and the data available. An IR system is more suitable to gathering information from a large collection of objects, without requiring a priori knowledge of their schemata or content.
In this work, we explore a new way of merging the two paradigms for use in ad-hoc analytical querying and self-service BI. We developed a novel type of hybrid DBMS/IR system called *DrillBeyond*, that blurs the line between these system, in all three aspects mentioned above: type of data managed, query language used, and nature of the query result. DrillBeyond processes so called *open world SQL queries*, a mix of relational queries on a local database with keyword-based lookup of additional web data sources that are then automatically joined and processed in one query. Consider a simple exemplary scenario using the TPC-H schema[1], in which a user analyses sales per country. Now consider a case in which the user wants to limit the scope of his or her analysis to those countries with a high gross domestic product, or

SSDBM '15, June 29 - July 01, 2015, La Jolla, CA, USA
© 2015 Copyright held by the owner/author(s). Publication rights licensed to ACM. ISBN 978-1-4503-3709-0/15/06...$15.00
DOI: http://dx.doi.org/10.1145/2791347.2791370

Listing 1 Running Example: Open world SQL query with added attribute highlighted

```
select n_name, gdp , avg(o_totalprice)
from nation, customer, orders
where
    n_nationkey=c_nationkey
    andc_custkey=o_custkey
    and gdp > 10.0
group by n_name, gdp
order by gdp desc
```

GDP. In a typical warehouse scenario, this involves manually searching a dataset that contains information about GDPs, moving it through an ETL process, reformulating the query to contain the predicate and an additional join with the imported table and finally rerunning the query. Additionally, if the user is not content with the results using this particular external source, he or she may have to go through the search and integration process iteratively until a suitable data source has been found.

For a situational one-of analysis query, this effort could be reduced if the DBMS' *query language* would directly support looking up and integrating web data sources as part of its query processing, and allow the specification of such queries directly in SQL, as shown in Figure 1. Note that the attribute *GDP* is not part of the TPC-H schema, and is specified only using a keyword embedded in the SQL query. The basic problem with processing this query is to retrieve the missing attribute, given the entities in the local database, i.e., the countries in the `Nation` table, from external data sources. Recent database research has explored the use of structured data extracted from the web to augment given entities with additional attributes, a process sometimes called *Entity Augmentation*[10]. While there is related work on processing entity augmentation queries themselves (see Section 6), the aspect of integrating entity augmentation queries into regular relational query processing has not been discussed previously. Existing literature stresses how automatic enrichment of user data using web data can help the user make the most of his or her data, but only considers entity augmentation on an isolated table, as shown in Figure 1a. Data analysis tasks however are often based on RDBMS, and the entity augmentation is not performed in isolation but in the context of complex SQL queries. DrillBeyond processes such queries by means of a web data-based *entity augmentation* operator as part of its query plans, as illustrated in Figure 1b. This operator encapsulates an entity augmentation query as a subquery of a relational query. Our proposed system thus utilizes two *types of data*: both a local relational database as well as an index of web data sources, i.e., web tables.

Finally, as we noted above, since the attributes are augmented from a large corpus of heterogeneous web sources, a user may want to consider several alternative sources. In the example above, the attribute was specified as just the keyword *"gdp"*. However, the real world concept is more complex, with many variants such as nominal GDP or GDP based on purchase power parity, different years of validity and different sources. A user may not even know on the spot which variant he or she is interested in. Furthermore the automatic matching system used to match web sources with the local data, no matter how sophisticated, introduces the additional uncertainty of possible matching errors.

Information retrieval systems solve this problem of uncertainty in sources and unclear user intent by presenting not one exact, but multiple, ranked, alternative answers.

For this reason, DrillBeyond answers open world SQL queries with multiple alternative query results, each based on a different external data source, or set of data sources, and presents the results as ranked alternatives to the user. It therefore produces structured results of exactly the form specified by the user query, just as a regular DBMS, but also presents several possible versions of the result, similarly to an information retrieval system.

In this paper, we describe the design of our hybrid DB/IR system and discuss issues related to integrating an information retrieval style operator into an relational database management system. In particular, our contributions are the following:

• We designed the *DrillBeyond* system and its entity augmentation operator, which implements top-k entity augmentation based on web tables as part of regular query processing.
• We discuss issues regarding placement of this novel operator with respect to runtime and answer quality, and propose a cost model as well as plan- and run-time optimization rules.
• We detail how to efficiently process a query multiple times based on different candidate data sources.
• We explore how the entity augmentation process itself can be improved through information available in the context of an SQL query.
• We created a practical implementation of the operator and our optimizations in PostgreSQL, which allows us to meaningfully evaluate the operator on standard test databases and fully-featured SQL queries, both regarding runtime and influence of the techniques discussed above.

Note that we presented an earlier demo version of DrillBeyond, focusing on the query model and GUI in [4]. A video of this original demonstration is still available[1].

2. OPEN WORLD SQL QUERIES

In this Section we define the main concept used in this paper, Entity Augmentation Queries and Open World SQL Queries, give the semantics of these query types, and discuss the technical challenges that arise when integrating them into an RDBMS.

2.1 Top-K Entity Augmentation Queries

So called *Entity Augmentation Queries* (EAQ) are of special interest for data analysis. This type of query is defined by an existing table and an attribute not defined for this table, the answer consisting of this attribute's values for entities in the table retrieved from external sources: $Q_{EA}(a_+, R(a_1,...,a_n)) = R(a_+)$. Here, a_+ denotes the attribute to be augmented and R the existing relation with attributes $a_1,...a_n$. An *Entity Augmentation System* (EAS), is used to answer this type of query. It is responsible for looking up data sources on the web, matching them to the existing table, and possibly for merging multiple candidate sources into one result. These properties make this type of

[1]https://wwwdb.inf.tu-dresden.de/edyra/DrillBeyond

querying both powerful as well as user-friendly, and thus interesting for ad-hoc analysis queries.
We discuss this type of query as a subquery of a regular SQL query. Our definition therefore includes an additional set of *hints* H to the entity augmentation system, which are extracted from the context given by the outer SQL query: $Q_{EA}(a_+, R(a_1, ..., a_n), H)$. These hints are used to guide the search and the ranking of web data sources, which we discuss in Section 3.4.
Finally, as discussed in Section 1, we argue that users should be able to choose from a ranked list of possible query results, as they would with web search. This means that for a given tuple $t \in R$ the system will return a set of k possible augmented relations: $Q_{EA}(a_+, R, H) = \{R(a_+)_1, ...R(a_+)_k\}$. We do not elaborate on the methods we use to perform the actual entity augmentation process and the creation of the k augmentation solutions, but refer the reader to [5]. For this paper, we assume that a system with the interface defined above is available, and discuss only the influence SQL query context on the entity augmentation result.

2.2 Entity Augmentation in SQL queries

As discussed in Section 1, we aim at processing SQL queries that reference arbitrary additional attributes not defined in the schema. Note that we require the attribute to be only specified via a keyword and a local relation using standard SQL notation, e.g. "nation.population", with no specific information regarding data type, or join paths with the local data. A baseline approach would require the user to manually retrieve possible data sources for these attributes and apply existing integration methods to the existing base relations. Then a standard SQL query over the extended local schema could be run.
With DrillBeyond, we want to avoid an explicit data retrieval and integration step, by associating values of these additional attributes to instances at query processing time. This is achieved by running one or more entity augmentation queries at runtime, which can be seen as a new type of subquery of a regular relational query. Since an entity augmentation query will return multiple possible augmentations, an Open World SQL query will, instead of returning of definitive query answer, return multiple variants of the query result, each based on different data sources. Additionally we will show that incorporating entity augmentation directly into relational query processing can also improve both accuracy of the augmentation, as well as runtime performance.

2.3 System Integration Challenges

In this section, we identify the main technical challenges that arise when integrating top-k entity augmentation with regular relational query processing. We discuss two major challenging areas: *multi-solution query processing* and *open world query planning.*
Multi-solution Query Processing: Recall that we are aiming to produce multiple query result versions for a single user query, each based on different possible augmentations. The naive way to implement this is to process the query multiple times, i.e., if k possible augmentations are to be used, the runtime of the SQL query is increased by this factor k. This is not acceptable, since it is likely that in many open world SQL queries, the majority of the processing time will still be spent processing local data, which does not change between runs of the query. For example, consider

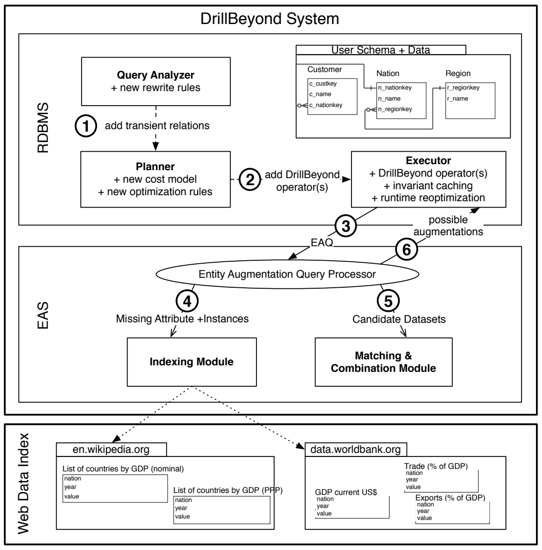


Figure 2: System architecture and high level control flow

again the example open world SQL depicted in Figure 1. Here, a large part of the work consists of local joins between the relations `Customer` and `Order`, and aggregation of the local attribute `o_totalprice`. When processing this query multiple times based on different augmentations, only the set of nations that pass the predicate on `gdp`, as well as the order of result tuples would change, but not the aggregates for the individual nations.
Consequently, the a major challenge in including top-k entity augmentation into relational query processing is to plan and process a queries in a way that minimizes duplicate work between query variant executions.
Open World Query Planing: The second challenge arises from the fact that properties of the joined web tables are not fully known at plan-time, since they are determined by the set of entities running through the DrillBeyond operator at run-time. For instance, estimating the selectivity of a predicate over an open attribute is not easy, as the set of sources that will be used is not known at planning-time. The same holds for determining the open attributes' metadata, such as the data type, since we do not require the user to specify it in the query. The question is thus how to plan queries with relations only known at run-time.

3. THE DRILLBEYOND SYSTEM

To solve these challenges and enable entity augmentation queries as part of relational query processing, we designed the DrillBeyond system. The next sections will detail the required changes to the RDBMS architecture and query processing.

3.1 System Architecture

Processing open world SQL queries requires an entity augmentation system, a web data index system, and modifications to three existing RDBMS components: the analyzer, the planner and the executor. Figure 2 gives an overview of the changed and new components, and also includes a high level description of the changes in control flow.

• **Web Data Index**: Most noticeably, an index of available web data sources is necessary. Typical entity augmentation methods based on web tables assume that a large scale corpus of web tables extracted from a web crawl is available. DrillBeyond does not have special requirements regarding this index and the system that its managed by: A generic system that exposes an interface for keyword-based document search, e.g. Solr or ElasticSearch, is sufficient.
• **Entity Augmentation Subsystem**: This component implements the entity augmentation processing, exposing an interface as discussed in Section 2.1. It interfaces with the index system to retrieve web data sources, e.g., web tables, and the core RDBMS components to provide augmentation services to the executor and the planner.
• **Query Analyzer**: The first step in DrillBeyond query processing is triggered by the query analyzer, which maps tokens in the SQL query to objects in the database's metadata catalog. Unrecognized tokens, such as *gdp* in the running example (Figure 1), lead to an error in a typical RDBMS. In the DrillBeyond system, we take a minimally invasive approach: we introduce transient metadata for the duration of the query, so that the regular analysis can continue. The query is then rewritten to include an additional join with a transient relation, effectively introducing a source for the missing attribute into the query processing, and also paving the way for the DrillBeyond operator to be placed by the regular join order planning mechanisms of the DBMS.
• **Query Planner**: Multi-solution processing requires plans that minimize the overhead of creating multiple result variants, while the lack of plan-time knowledge about the data sources requires plan adaption. We detail our approach to these problems in Section 4.
• **Executor**: The executor implements the repeated execution of operator trees to create the top-k query result, using the DrillBeyond operator, which we will detail in the next section.

3.2 The DrillBeyond Operator

In its basic form, the DrillBeyond plan operator, denoted ω, is designed to resemble a join operator, to facilitate integration with the existing system architecture. Note however, that in contrast to a regular join, only one of the joined tables is known at plan-time, while the other table, as well as the join attributes are decided at run-time. These run-time decisions are made by the entity augmentation system based on the input tuples of the operator. Specifically, the operator extracts distinct combinations of textual attributes from input tuples, as these are used to functionally determine the values of the augmented attribute. In the example query shown in Figure 1, the set of `Nation` tuples that reach the operator determine the input to the entity augmentation subquery, and therefore determine the web data sources that are are retrieved.
We will now consider the implementation specifics of the operator. Algorithm 1 shows the specifics of the state kept in the operator and the implementations of its iterator interface and helper functions. The traditional interface functions *Init()*, *Next()* and *ReScan()* are called by the DBMS during regular query processing. The first, called before executing the query the first time, initializes state, specifically a tuplestore for materializing the lower operator's output, a hash table mapping local textual attribute values to augmented values, and two variables *state* and *n*, determining the behaviour of the operator when *Next()* is called.

Algorithm 1 DrillBeyond operator

```
function INIT
    $state \leftarrow$ ‘collecting’
    $tuplestore \leftarrow \emptyset$
    $augMap \leftarrow HashMap()$
    $n \leftarrow 0$                                        ▷ Iterations

function NEXT
    if $state =$ ‘collecting’ then
        COLLECT()
        AUGMENT()
        $state \leftarrow$ ‘projecting’
    return PROJECT()

function COLLECT
    while $true$ do                              ▷ Retrieve all tuples
        $t \leftarrow$ NEXT($childPlan$)
        if $t = NULL$ then
            break
        $tuplestore \leftarrow t$
        $augKey \leftarrow$ TEXTATTRS($t$)
        if $augKey \notin augMap$ then
            $augMap[augKey] \leftarrow \emptyset$

function AUGMENT
    $augReq \leftarrow (\forall k \in augMap \mid augMap[k] = \emptyset)$
    for all $augKey, [augValues...] \in$ SEND($augReq$) do
        $augMap[augKey] \leftarrow [augValues...]$

function PROJECT
    $t \leftarrow$ NEXT($tuplestore$)
    if $t = NULL$ then return $NULL$
    $augKey \leftarrow$ TEXTATTRS($t$)
    $t[augAttr] = augMap[augKey][n]$ return $t$

function RESCAN
    $state \leftarrow$ ‘collecting’
    $tuplestore \leftarrow \emptyset$

function NEXTVARIANT
    RESCAN($tuplestore$)
    $n \leftarrow n + 1$
```

The *Next()* function produces augmented tuples. This is done in three phases: *Collect()*, *Augment()* and *Project()*. On the first call to *Next()*, since no augmented values are available, the first two phases are triggered. In the *Collect()* phase, the operator pulls and stores all tuples that the lower plan operator can product, making DrillBeyond a blocking operator. The reasons for blocking are discussed in Section 3.3. In this phase, the operator also stores the textual attributes originating from the augmented relation and its context in a hash table, to obtain all distinct combinations of textual values in the input tuples. The concept of augmentation context is discussed in Section 3.4.
In the *Augment()* phase, all entries in the augmentation map that do not yet have values associated with them are passed to augmentation system as one augmentation context. After successfully retrieving values for all collected tuples, the operator is put into the *projecting* state, and produces the first output tuple. Output tuples are produced in the *Project()* function by replaying the stored tuples and filling the augmentation attribute by looking up values in the hash table.
The *ReScan()* function is called by the DBMS executor when subtrees have to be re-executed, e.g., in dependent subqueries

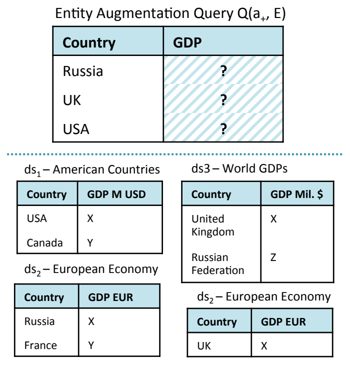
or below a nested loop join. Here, the operator empties its tuplestore and changes state to collect new input, but keeps its augmentation hash table, to prevent expensive re-augmentation for values that have already been seen. Finally, the *NextVariant()* is an interface extension not seen in typical RDBMS operators which is necessary for producing the multi-variant query results as discussed in Section 2. Instead of dropping the tuplestore, it just calls *ReScan()* on it and increases its internal iteration counter n. This makes sure that in a new execution of the query plan, operators below the DrillBeyond operator are not called again, but instead their materialized output will be replayed, and augmented with the next augmentation variant in the *Project()* function. Having discussed the basic operator functionality, we will now discuss more advanced questions in the next subsections: why is the operator blocking, where should it be placed in the query plan, what is its runtime cost and how can it be optimized?

3.3 Augmentation Granularity

A naive entity augmentation operator would work *tuple-at-a-time*, i.e., hand each tuple to the augmentation system as it receives it. This way, it would be most compatible to the iterator-based query processing used in most traditional RDBMS operators, and not need to block. However, augmenting each tuple on its own implies looking up and matching web data sources for each tuple on its own. Consider the simplified augmentation example shown in Figure 3, where the table to be augmented is on top, and the available web data sources at the bottom. With the *tuple-at-a-time* style, the augmentation system may choose ds_1, ds_2 and ds_4 for the USA, Russia and UK tuples respectively. These sources match each individual tuple best, and the individual tuples are what the augmentation system can process in this case.
Still, we can see that the augmentation system can not perform optimally with regard to the query as a whole, as it is not provided with the overall query context. While the chosen sources are the best fitting for each individual tuple, they do not form a consistent answer together, as the units of currency do not match. If the augmentation system is instead provided with the set the complete set of tuples as the input for one augmentation query, the more consistent solution comprised of ds_1 and ds_3 can be constructed, even though the individual entity matches are slightly worse.
We conclude that for reasons of result quality the DrillBeyond operator needs to be a blocking operator. This means it consumes tuples from underlying operators until these are exhausted, then hands them over to the augmentation system, and produces the first result tuple only when it returns. Referring back to Algorithm 1, this blocking behavior is realized in the state "collecting" in function *Next()*.

3.4 Context-dependent Results

Having established the DrillBeyond operator as blocking, we will now consider the question of where to place it in a query plan. We noted in Section 3.3 that the augmentation result is dependent on the set of input tuples to the augmentation system. However, this is not only affected by the augmentation granularity, but also the position of the operator in the graph. For example, consider again the query shown in Figure 1, and two possible query plans shown in Figure 4, where ω depicts the DrillBeyond operator. The


Entity Augmentation Query Q(a_+, E)

Country	GDP
Russia	?
UK	?
USA	?

ds_1 – American Countries

Country	GDP M USD
USA	X
Canada	Y

ds3 – World GDPs

Country	GDP Mil. $
United Kingdom	X
Russian Federation	Z

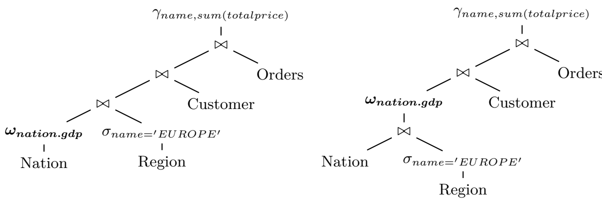
ds_2 – European Economy

Country	GDP EUR
Russia	X
France	Y

ds_2 – European Economy

Country	GDP EUR
UK	X

Figure 3: Example augmentation problem



(a) DrillBeyond first (b) Regular join first

Figure 4: Possible drillbeyond operator placements

set of `Nation`-tuples that reach the operator determine the input to the entity augmentation subquery, and therefore determine the web data sources that are are retrieved. In variant (a), the DrillBeyond operator is executed directly after the scan of the `Nation` table, while in (b) it is executed after the local join and filter with the `Region` table has been performed. In the first case, the entity augmentation subquery will retrieve, match, combine and rank datasets for all countries in the local database. Likely, these will be datasets about the countries of the world. In the second case, the local join will remove tuples about non-European nations, so the results of the entity augmentation will be more likely be based on data sources specifically about Europe. Furthermore, completing the join with the `Region` table does not only limit the scope of the query, it also adds context to each tuple by adding `Region`'s attributes. So even if the join would not act as a filter limiting the number of tuples, adding information about the region name will improve the accuracy of the augmentation system. For example, while augmenting a set of City-tuples may be hard and error-prone because of the ambiguity of common city names such as "Springfield", augmenting after a join with a *State* will be a more realistic task for the augmentation system.
Therefore, for the DrillBeyond operator ω we can see that we can see that $\omega_{R,a_+}(\sigma(R)) \neq \sigma(R)(\omega_{R,a_+})$. We therefore define the following placement rule that gives a lower bound on the placement of the operator.
• **Lower Placement Bound**: The DrillBeyond operator augmenting a relation R can only be placed when all filters on R, e.g., joins and predicates, have been applied.
In other words, DrillBeyond is always applied to the minimum number of distinct combinations of textual column values in the tuples of R, as these determine the matching process and its result.

3.5 Pushing SQL query context

As mentioned in Section 2.1, context from the outer SQL query can be used to improve the accuracy and runtime of the inner entity augmentation query, when compared to isolated augmentation. We already discussed also filter effects through joins and predicates in Section 3.4. These filters work implicitly to improve the augmentation quality by narrowing the scope of the augmentation operation, and do not require any changes to the augmentation system or its API. However, we can further improve the augmentation by explicitly pushing additional query knowledge to augmentation system. Specifically, we push two types of information: *type information* and *predicates on augmented attributes*.

• **Type Information:** Though the user can specify a data type such as *text* or *double* for open attributes using SQL syntax, we do not expect those annotations to be provided. However, in many cases it is possible to infer the type of the open attributes from the surrounding query by applying methods of type inference to SQL. We can pass type information to the augmentation system, which in turn uses this type information to restrict the set of candidate data sources to those matching the open attribute's type.

Consider again the exemplary query shown in Figure 1. From the constraint on the *gdp* attribute it can be inferred that it must be of a numeric type. This allows the augmentation system both to reduce its runtime and increase precision by pruning non-numeric candidate sources.

• **Predicates on augmented attributes:** In addition to the data type of open attributes, we can also push the predicates on open attributes themselves down to the augmentation system. This allows a similar, but more sophisticated, candidate pruning. We assume that users expect some filtering on the database instance level to happen when specifying a predicate, i.e., we assume that some domain-knowledge is encoded in the predicate.

Consider again the exemplary query shown in Figure 1. The user clearly intends to filter low *GDP* countries, and has given a large integer number as the specific condition. Though the user query is given only with the very general keyword *GDP*, by also considering the predicate, the augmentation system can discard all candidate datasets that give the GDP as a percentage or rank, because those will not discriminate the entities with respect to the predicate. In addition to all-or-nothing filtering, we can use the predicate to improve the augmentation system's ranking function. Specifically, we can check how well the data source's values fit to the predicate that will be applied, i.e., how well the data source discriminates the entities with respect to the predicate. Using the SQL context, we can thus add the following sub-score to the scoring functions already in place in our augmentation system REA (see Section 5.1), given a predicate p and a candidate data source C:

$$s_p(p, C) = \begin{cases} 0 & \text{if } \neg applies(p, C) \\ \frac{\sum_{v \in C} 1.0 - |\log_{10} p - \log_{10} v|}{|C|} & \text{if numeric(p)} \\ 1.0 - (2 * |0.5 - sel(p, C)|) & \text{otherwise} \end{cases} \quad (1)$$

Note that we also include a special rule for numeric predicates: we check whether the predicate value and the average of the data source's values are in the same order or magnitude. We will show the improvements in precision and runtime of the augmentation system when applying this score in Section 5.

3.6 Cost Model & Initial Placement Strategy

As mentioned, the operator is modeled to resemble a join from the perspective of the DBMS. We can therefore reuse the existing join optimization machinery to place the DrillBeyond operator. This however depends on a model for the operator runtime cost and the operator output cardinality.

• **Output Cardinality**: In the most basic case, the operator produces exactly as many tuples as its input relation, as it just adds a single attributes. However, we also must consider selectivity of possible predicates on augmented attributes. Since at plan-time we do not assume any knowledge about which web data sources will be used to augment an attribute, correctly estimating selectivity is almost impossible. We therefore initially use the DBMS default selectivities for different types of predicates, and employ run-time optimizations to compensate when more information is available (see Section 4).

• **Cost Model**: The operators runtime depends on three components: The first part is incoming tuple processing, the second entity augmentation, and the third is projecting tuples with the augmented attribute (see Algorithm 1). We can estimate the cost of the operator using a similar model as for a hash join, as phases one and three, hashing the child relation, and then probing it against the augmented hash table correspond to the phases of a hash join. Additionally, there is the cost of phase two, the actual augmentation, which depends not on the number of tuples, but on the number of distinct entries in the augmentation hash table. The processing cost per entry depends on the augmentation algorithms, and is therefore not easy to model in the context of a generic DBMS cost model. For our cost model we therefore assume an additional large constant factor $\mathcal{C}$ for the phase two component which, for evaluation purposes, we learn from previous executions of our augmentation system (see Section 5).

$$Cost_{Drb} = 2 * |subPlan| + \mathcal{C} * |distinct(subPlan)| \quad (2)$$

We use the cost model together with the lower placement bound rule introduced in Section 3.4 to place the operator in the query plan. The rule is enforced by pruning join orders in which the operator would be placed before all filters to the augmentation base relation are applied.

4. PROCESSING MULTI-RESULT QUERIES

So far, we have considered the DrillBeyond operator in a single query result setting. However, as discussed in Section 2.2, we aim at translating the top-k result returned by our augmentation system into a top-k SQL result.

The naive method, given our operator, is to simply re-execute the query plan k times, and after each execution trigger the projection of a new augmentation result from the operator via the *NextVariant()* API depicted in Algorithm 1. This obviously leads to duplicated work , as only the output of the DrillBeyond operator changes between executions, while the other parts of the query plan operate the same. Consider the query plan in Figure 5. Here, only the values of the `gdp` attribute would change between query runs, while the operations, notably the more expensive joins with the `Customer` and `Orders` relations and the grouping would not change. This means that simple re-execution would increase time to compute the query k-fold, with most of the work

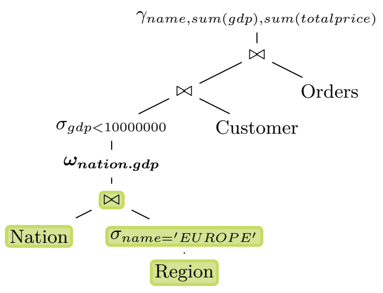


Figure 5: Plan Invariants

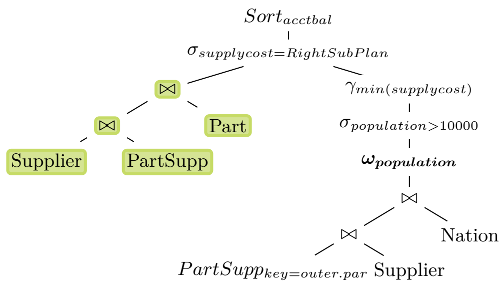

$Sort_{acctbal}$
$\sigma_{supplycost=RightSubPlan}$
Part
Supplier
PartSupp
$\gamma_{min(supplycost)}$
$\sigma_{population>10000}$
$\omega_{population}$
Nation
$PartSupp_{key=outer.par}$
Supplier

Figure 6: Invariant Subtrees

being inefficient duplicate work. Our first approach to this problem is to identify, and then maximize invariant parts of the multiple executions, and then prevent their re-execution through materializing intermediate results. As a first observation, note that all tuple flows below the DrillBeyond operator can actually never change between executions: in the example shown in Figure 5, these operators are underlined. The cost of those operations can be minimized by materializing the input to the DrillBeyond operator. This elementary optimization is is already included in the basic operator implementation shown in Algorithm 1 in the form of the tuple store created by the operator. The changing augmentation output may, however, also influence the result of other operators further up the query plan. In the example, the aggregation of the `gdp` attribute will change its output between execution. Note however, that even the aggregation of the regular attribute `totalprice` is influenced by the changing `gdp` values: since the selectivity of the predicate on `gdp` may be different in different data sources, the set of `Orders` tuples that has to be aggregated may also change between executions.
The next sections will introduce our optimization strategies for these problem strategies: invariant caching, augmentation operator splitting, selection pull-up, partial selection, and finally, run-time reoptimization.

4.1 Invariant Caching

We already discussed the necessity for materializing intermediate results, and introduced the basic caching done by the DrillBeyond operator itself. However, this is not the only opportunity for caching. Consider the following query plan in Figure 6, modeled after TPCH query 2, in which a dependent scalar subquery (on the right side) is executed for each tuple of the outer query. A DrillBeyond operator

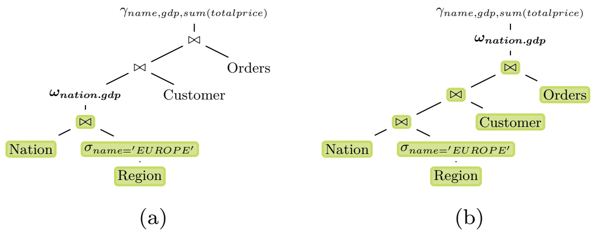


Figure 7: Influence of DrillBeyond operator placement

and a predicate on the augmented attribute was added in the dependent subquery. Here, the result of the main query before the selection is invariant between query results, as the augmented values only change the result of the filtering subquery.
In general, operators that have no DrillBeyond operator in their descendants are invariant.We exploit this using a planning pass, that inserts materialization nodes into the plan. The idea is to recursively descend into the query plan, until a DrillBeyond operator is encountered, which marks all nodes above it as varying. A node is marked varying if any of its child subtrees or any subqueries evaluated at the node are marked varying.
At each varying node, we check whether explicitly materializing any subtree would be beneficial. A good example would be intermediate join results, such as the whole left subtree underneath the top selection in Figure 6. Here, the addition of an explicit materialization node above this subtree would speedup further executions of the query plan considerably. Note, that even a subtree without DrillBeyond operators may be varying, if it contains parameters, e.g., the inner side of a nested loop join. In this case, the variability of a sibling subtree will propagate, as the parameters may change between executions if the other subtree is varying.
Having introduced our methods for invariant caching as a first step to optimize multi-result queries, the limits of this first method have also become clear: we can only cache invariant subtrees, and depending on the cost based placement of the DrillBeyond operator, most of the query may be varying and thus not amendable to caching. We will therefore introduce further optimizations in next sections.

4.2 Separating Augmentation Input and Output

So far we have optimized by caching invariant parts of the query plan as it is produced by the planner. However, our planning process as described in Section 3.2 assures only that the overall order of joins and DrillBeyond operators is optimal with respect to the cost model, and correct with respect the lower placement bound. It does not however, try to maximize the invariant parts of the query. Consider the following example queries, shown in Figure 7. The first case (Figure 7a) is optimal with respect to a single query execution: the operator is applied to the smallest possible intermediate result, while the lower placement bound, the join with `Region`, is adhered to. Still, with respect to multiple execution and the caching mechanisms described above, the second plan (Figure 7b) is more efficient. By moving ω up the tree, this plan trades higher cost for executing the DrillBeyond operator once for a much larger invariant

subtree, which pays off over more executions. While it would be possible to use an extended cost model that incorporates the costs for multiple plan executions and thus place choose the second plan, we can do even better.
The key observation is, that to maximize the invariant parts of the query plan, the operator should be placed as late as possible, i.e., not earlier than the augmented values need to be accessed. This is the upper bound to the ω operator placement. Though a seemingly obvious observation, it enables a useful optimization: we can separate the input part of the operator from the actual projection of augmented values.
We split the DrillBeyond operator in two parts, one performing the augmentation, symbolized with ω, and one doing the projection of values, depicted as Ω. The ω operator performs the input hashing and triggers the augmentation process, i.e., it works through the *collect()* and *augment()* phases of the basic operator implementation shown in Figure 3.2. However, instead of projecting the actual augmented values, it projects placeholders, which can be uniquely mapped to the actual value arrays returned by the augmentation system. Technically, these can be implemented as pointers into the *augMap* hash table that is populated in the *augment()* phase. These placeholders traverse the operator graph until the Ω operator, which dereferences to the correct value array, and projects a single augmented value depending on the current query iteration, i.e., the current n in Algorithm 1. We place Ω at exactly the upper placement bound defined above, and ω above the lower placement bound, minimizing augmentation operator cost and maximizing the size of the invariant query parts.
This optimization provides more efficiency gains, but is still limited by the first access to the augmented attribute in the query plan, i.e., its effect is based on the distance between ω and Ω. We will therefore explore more optimizations that delay the first access to an augmented attribute: *selection pull-up* and *partial selection.*

4.3 Selection Pull-Up

The strategy of projecting augmented attributes as late as possible is in conflict with one of the more fundamental classic optimizations, namely selection push-down. For example, if a predicate on the augmented attribute is given, it would make sense to evaluate this predicate as soon as possible, i.e., directly after ω, to limit the size of intermediate results, as done in traditional query optimization., i.e., $\ldots \bowtie (\ldots \bowtie \sigma_{a_+}(\omega_{a_+}(R))$. This however means that our ω / Ω split cannot be applied anymore, since augmentation and first access to the values coincidence, i.e., we would have $\ldots \bowtie (\ldots \bowtie \sigma_{a_+}(\Omega_{a_+}(\omega_{a_+}(R)))$, eliminating all benefit of the split. We therefore have to decide between two alternatives, depicted in Figure 8: traditional selection push-down at the cost of having more varying nodes in the plan, or the opposite approach: *selection pull-up.* With this method, we place Ω as described in Section 4.2 while ignoring predicates on the augmented attribute, i.e., before the first other access. The trade-off in this case is between smaller intermediate join results with the plan in Figure 8a and larger but invariant joins in Figure 8b. Which plan is to be favored depends on the selectivity of the predicate on the augmented attribute. We thus employ a cost-based approach to decide on whether to use selection pull-up. Two questions are to be answered: what cost model to base this decision on, and how to estimate the selectivity of the predicate. Concerning the cost-model: since we have a binary decision we only need to compare the results of two suitable models and pick the cheaper plan. We use the following two models for the cost of the subtree above the ω:

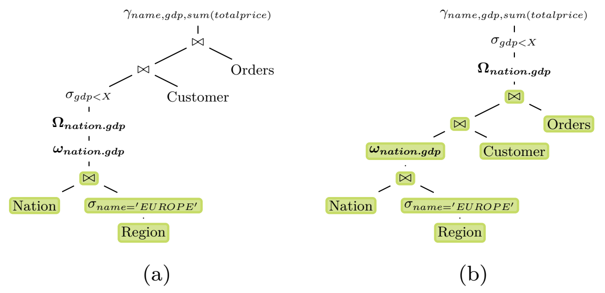

$\gamma_{name,gdp,sum(totalprice)}$
$\bowtie$
Orders
Customer
$\sigma_{gdp<X}$
$\boldsymbol{\Omega_{nation.gdp}}$
$\boldsymbol{\omega_{nation.gdp}}$
Nation
$\sigma_{name='EUROPE'}$
Region
(a)
(b)

Figure 8: Selection Pull-up and influence on invariants

$$C_k(\omega, \top) = \begin{cases} C_1(\omega, \Omega) + \sum_{i=1}^{k} s_i \cdot C_1(\Omega, \top) & \textit{(Pull-Up)} \\ \sum_{i=1}^{k} s_i \cdot C_1(\omega, \top) & \textit{(no Pull-Up)} \end{cases}$$

Here, C_k is the cost of k plan executions, s_i the selectivity of the i-th data source for the attribute in question, and $C(x, y)$ denoting the cost of the subplan from operator a until to operator b. With selection pull-up, we have to pay the price for the subplan between ω and Ω only one time, but with selectivity 1.0, as no predicate is applied. The rest of the plan, from Ω to the result, has to be executed k-times, but with the predicate applied. Without pull-up, the selection is applied early, but the whole plan is executed k-times.
The cost of the partial queries can be calculated from the normal query optimizer output, the k is user-defined and fixed, but what about the selectivities s_i?
As discussed in Section 2, we have no knowledge at plan-time about which data sources will be used, and thus can not easily estimate their selectivities. Instead of deciding the selection placement at plan-time when selectivities are hard to estimate, we therefore make the decision at run-time, specifically after the augmentation system has returned, i.e., in the *augment()* phase shown in Algorithm 1.
With the selectivities known, the ω operator decides using the cost-models defined above whether to project real values and perform the selection, in which case Ω operator becomes a dummy, and further executions must recalculate all operators in the (ω, Ω) range, or whether the selection is pulled up, and it emits only placeholders as described in Section 4.2.

4.4 Partial Selection

In the previous section, we discussed the trade-offs between intermediate result cardinalities and plan invariability that result from position of selection operators. In this discussion, pulling or pushing the selection was a binary decision: either execute the selection at ω, or at the corresponding Ω operator. This assumed that the web data sources used for the different augmentations are completely independent, i.e, that they will always lead to varying intermediate results. However, for many augmentation queries, the sources may actually agree on whether a predicate holds for a certain tuple. We can therefore introduce a *partial selection* operation, which selects all tuples for which the selection predicate holds in any of the available augmentations.

$$s_p(R) = \{t \in R \mid \exists i \in (1..k)\ P(t[a_{+_i}]))\}$$

Here, a_{+_i} denotes the augmented attribute in its i-th possible variant. The selectivity of this operation depends entirely on the correlation between the sources with respect to the predicate. It may range from being equal to the selectivity of the individual sources if they completely correlate, to being 1.0 if the predicate holds for every tuples in at least one variant.
Even though the effect varies, it can be lead to better query performance because we may perform partial selection at the ω operator, while still returning an invariant result. It can therefore be combined with selection pull-up described in Section 4.3, by returning placeholder values for tuples that pass the partial selection, and performing the data source-dependent final selection at the Ω operator.

4.5 Runtime Reoptimization

The optimizations proposed so far where concerned with optimizing the plan regarding invariants and selectivities. Dynamic selection pull-up even allows to adapt the plan at run-time, depending on the selectivity of predicates on augmented attributes. However, if the actual selectivity is very different than the one used for initial planning, there may be more optimization potential than just the placement of the selection operation. For example, the original choice of access paths for base relations, e.g., index vs. table scan, may be suboptimal with respect to the actual selectivity.
We therefore use a form of adaptive query processing: When executing a query, DrillBeyond always starts execution at the subtrees below ω operators, to be certain about selectivities as early as possible. DrillBeyond then invokes the planner to create a new plan using the better selectivity estimate.
At this point, either the average selectivity of the k augmentations or the selectivity of the partial selection over all augmentations (Section 4.4) can be used as an estimate. This decision depends on whether selection pull-up will be used with the new plan, and can be made using the selection pull-up cost model (Section 4.3) . The already executed subtrees underneath the ω nodes are then merged into the new plan, to make sure that materialized intermediate results and especially the results of the augmentation operation are reused with the new plan.

5. EVALUATION

In this section we present the results of our experimental study on efficiency of open world SQL query processing in the DrillBeyond system. We study the performance of the DrillBeyond relational operator in general, and the influence of our optimizations in particular: intermediate result caching, the ω/Ω split, selection pull-up and partial selection and run-time reoptimization. We also study how the use of SQL query context, especially predicate push-down to the augmentation system, can improve both augmentation quality.

5.1 Experimental Setup

In Section 3 we introduced our system architecture consisting of three components: the actual modified RDBMS, an augmentation system and a web data source index. As mentioned above, the augmentation system and web data corpus used are previous work. Still, we introduce the other components as well, to give a complete picture of the experimental setup.

Implementation: We implemented the operator described in Section 3 and the optimizations described in Section 4 into the PostgreSQL open source RDBMS and made it available on GitHub[2]. The optimizations discussed in Section 4 can be turned on and off individually, to study their individual performance impact. The augmentation system we employ is the REA system presented in [5] also available open source[3]. It creates a top-k list of possible attribute augmentations for a set of entities, based on a list of candidate data sources. For this paper, it was extended to be able to accept predicates as query hints to guide the search for data sources, as described in Section 3.5. To provide a list of candidate sources for the augmentation, we use an index over our *Dresden Web Table Corpus* (DWTC)[4]. This corpus contains 122M Web tables extracted from the Common Crawl[5], a publicly available Web crawl.
Test Database: Throughout the paper we used TPCH-based examples. Most relations in TPCH, such as `Supplier` and `Part` carry only artificial identifiers, i.e., "supplier_1" or "part_1", and can therefore not be extended with external data due to their lack of real-world identity. We use two different ways to address this issue: First, for the pure performance experiments that test our query processing optimizations, we use an entity augmentation system that generates artificial data instead of looking it up on the web. This also has the advantage that we can exactly specify properties of the returned augmentations, such as selectivities and correlation between augmentations, to test various aspects of query processing.
Second, for the quality related experiments, we created a variation of TPC-H, which replaces generic identifiers with real-world entities. For example, the names in the `Supplier` relation are replaced by real company names crawled from the web.
For all experiments, we used TPC-H with scale factor 1.0.
Test queries: We focus on a subset of the TPC-H queries in which dimension tables are used, as fact tables such as `Lineitem` or `Orders` do not contain information about real world entities that could be augmented. We added arbitrary attributes to one of the dimensions, by adding a `where`-clause of the form "relation.X > Y" to each query. For the performance experiments, X and Y are arbitrary, since we use generated augmentation data with specific selectivities for these experiments. For the quality experiments we will give specific (relation, attribute, predicate)-tuples in the respective sections.
Parameters: The main parameters are k, the number of augmentations and thus the number of SQL results to be produced for a query, and s, the selectivity of the predicate on the augmented attribute. If not stated differently, the we perform each experiment for the each $k \in (1, 3, 5, 10)$ and ten different levels of selectivity s, ranging between 0.01 and 0.99.

5.2 Performance

With the experimental setup and parameters as described above, we measured five different configurations of DrillBeyond to test the optimizations introduced in Section 4. The configurations are:

[2]http://github.com/JulianEberius/DrillBeyond
[3]http://github.com/JulianEberius/REA
[4]http://wwwdb.inf.tu-dresden.de/misc/dwtc
[5]http://commoncrawl.org

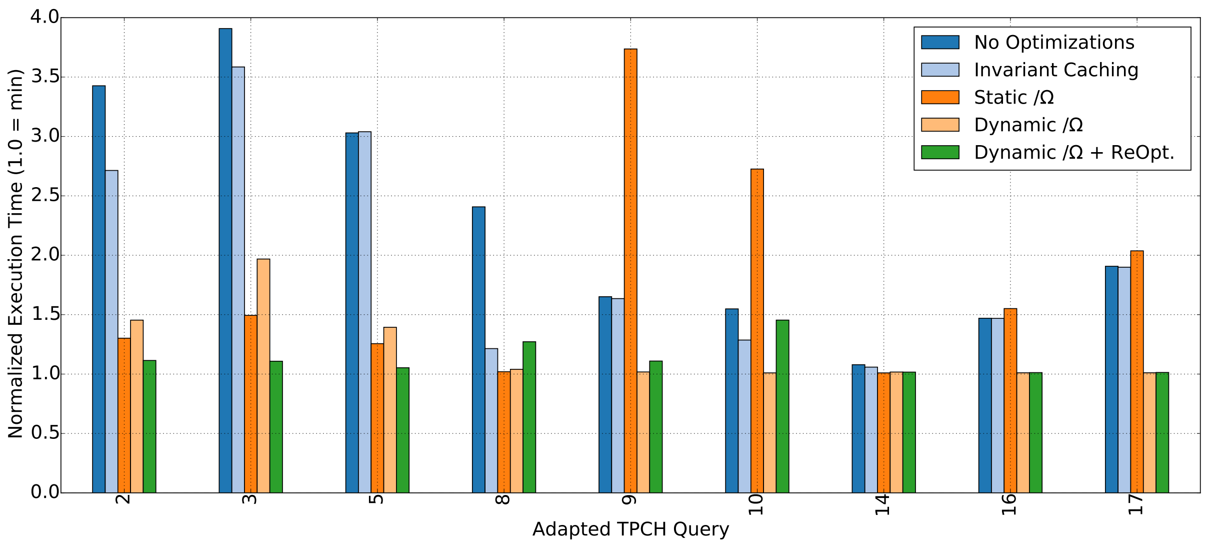


Figure 9: Overview of normalized execution times by query

• *No Optimization*: The query is planned using just the cost model introduced in Section 3.6 and default selectivity estimates. This means the query is planned for one execution, and not optmized for producing multiple results.
• *Invariant Caching*: In this configuration we enable the caching of invariant plan subtrees, as described in Section 4.1.
• *Static ω/Ω*: This configuration introduces the split between the augmentation operator ω using the cost model and the projection operator Ω placed at the first attribute usage, as introduced in Section 4.2. In this context, "static" means that the selection is always performed at the Ω operator, making most of the plan invariant, but also not utilizing the selectivity of predicates.
• *Dynamic ω/Ω*: This configuration introduces dynamic, runtime placement of the selection at either ω or Ω, as described in Section 4.3.
• *Dynamic ω/Ω + ReOpt.*: In this configuration, we enable the reoptimization of the remainder of the query plan after ω is executed, as introduced in Section 4.5.
Figure 9 shows the results using normalize execution runtime, i.e., all times are given as multiples of the fasted configuration in each individual measurement. The numbers in this overview are aggregated by query, i.e., aggregated over all of k and all s.
The first major observation is that the various optimization techniques respond quite differently to different queries. For example, invariant caching alone is almost good enough for Q8, the other techniques do not improve the significantly, runtime reoptimization even introduces a small overhead in this case. Still, in all other cases, invariant caching alone is not enough. Introducing the Ω operator with static selection improves most queries considerably, by making most of the plan invariant. However, some queries, such as Q9 and Q10, can profit so much from a selective predicate, that a static placement of the selection at Ω looses much optimization potential. In these cases, the basic invariant caching approach is even better, even though it re-executes more plan nodes, because it can execute the predicate directly at ω.
Focusing on the dynamic configurations, we can see that these improve even more queries, as they can dynamically

Scenario	Normalized Execution Time	Standard Deviation
No Optimizations	2.27	1.82
Invariant Caching	1.99	1.59
Static ω/Ω	1.79	2.35
Dynamic ω/Ω	1.21	0.67
Dynamic ω/Ω + Re-Opt.	1.13	0.28

Table 1: Normalized execution time, average and standard deviations

place the predicate execution depending on k, and even s. However, we can also see that introducing a new cost-based dynamic decision into query processing also introduces some new uncertainty. For example, while the dynamic selection approach without plan reoptimization produces the best plan in Q10, it is even worse than static selection in queries Q2, Q3 and Q5. Finally, run-time reoptimization, which should lead to the best query plans in theory, is not globally optimal either, as seen in Q10. However, in Table 1, we can see that while the reoptmization method does not gain much to the dynamic approach in average, its result show a considerably lower standard deviation, i.e., it produces more conservative plans.
To give an insight of how the observed aggregated effects arise, Figure 10 shows absolute execution times for a single query, Q9, plotted by selectivity of the predicate, and once for each $k \in (1, 3, 5, 10)$. We can clearly observe the difference between pushing or pulling the selection on the augmented attribute. In the invariant caching case, selection is performed early at ω, which is faster for low selectivities and low k. With more repeated query executions, e.g. for $k = 10$, the higher variability in the plan makes this strategy infeasible. In contrast, in the case of static selection at Ω, most of the plan becomes invariant, and thus repeated executions are not very costly. However, this strategy can not benefit from low selectivities. The dynamic switching combines the benefits of both variants, although the cost model is not perfect: selection pull-up is chosen slightly to

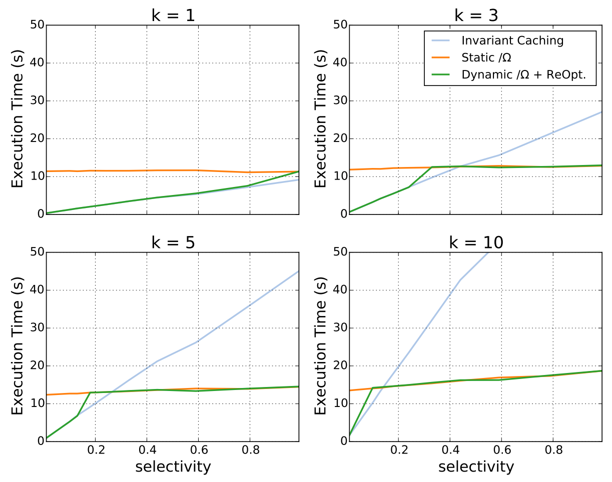
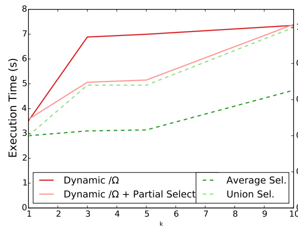
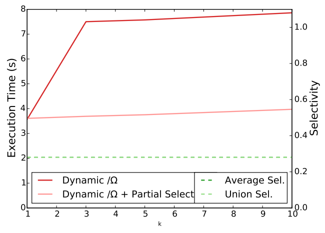


Figure 10: Execution time for query 9, by selectivity



(a) gdp > 60.0



(b) population > 1E8

Figure 11: Effects of partial selection

early in this query.
Finally, we want to study the effects of partial selection, as introduced in Section 4.4. As mentioned in Section 5.1, our main performance experiments have been carried out with generated augmentations, so that all levels of selectivity could be easily observed. Since partial selection depends mostly on the correlation between the data sources that the augmentations are based on, it is more interesting to study with augmentations using real web data. So for the next experiment, we used our web table retrieval & matching system REA (introduced in Section 5.1). We used one SQL query, similar to TPC-H query 5:

```
select n_name, <attr> , sum(...)
from nation, customer, orders, supplier, lineitem
where
    ...
    and <attr> <predicate>
group by n_name, <attr>
order by <attr> desc
```

We evaluated this query with two different (attribute, predicate)-pairs: (gdp, >60) and (population, >1E8), and the results are shown in Figure 11. In this figure, the x-axis shows k, the number of augmentations requests, while the first y-axis shows the execution time with and without partial selection. Finally, the second y-axis shows both the average selectivity over all augmentations, as well as the union selectivity as defined in Section 4.4.
The union selectivity can be understood as the ratio of tuples that pass the predicate in at least one of the augmentations. In the first case we can see the effect of the number of augmentations on union selectivity. Although the average selectivity rises slowly until 0.65, the union selectivity rises more quickly and reaches 1.0. This means that as k rises, even though in all augmentations only a subset of nations is selected, all nations are selected at least once.
However, as shown in Figure 11, as long as the union selectivity is below 1.0, the runtime can be improved proportionately using partial selection. Whether this is the case depends entirely on the correlation between data sources used. Consider Figure 11b, which shows the much less ambiguous query for "population". In this case, all augmentations agree on which countries have more than 100 million inhabitants, which leads to equal average and union selectivity. This allows to save a corresponding amount of work in query processing, as large parts of the database never need to be touched.

Concept	Attribute	Predicate
company	employees	>50000
company	founded	<1900
company	revenue	>100
company	revenue growth	>10.0
country	population	>1E8
country	population growth	>2.0

Table 2: Concepts, attributes and predicates used in precision evaluation

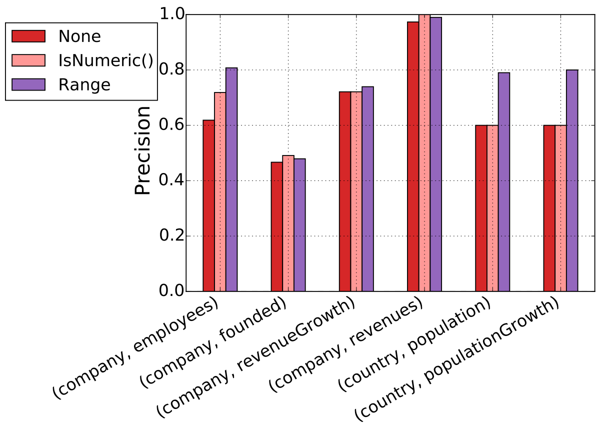


Figure 12: Changes in precision by type of predicate used

5.3 Augmentation Quality

For the next experiments, we again used the REA system, and extended it to handle predicates on augmentation attributes as guides for the web tables search and scoring process. Following our discussion in Section 3.5, we first added type information as a query hint, by implementing an "*IsNumeric*" predicate in REA. This allows DrillBeyond to signal REA that the augmented attribute should be numeric, which in turn allows REA to filter non-numeric web table columns from its search. In a second step, we allowed range predicates to be passed to REA, which uses the equation given in Section 3.5 to improve its data source ranking.
With this improved REA we ran a subset of the precision tests given in the original REA paper [5]. The set of concepts and attributes used, as well as the predicates we added to test predicate hints are shown in Table 2. Except for the added query hints, all of the precision evaluation setup was the same as in [5]. The results are shown in Figure 12. We can see that adding type information and range predicates

improves precision, although the changes are notable only in some domains. For example, in the "employees" domain, the effect was mostly due to filtering web tables that gave information about local branches of the queried companies, giving either no numbers, or low numbers in relation to the query predicate.

6. RELATED WORK

DrillBeyond build upon methods for automatic, web table-based entity augmentation. A notable first example of such work is [2], which described a set of basic operators that facilitate the integration of many structured data sources from the web. One of these operators, called `Extend`, attempts to find matching Web tables for a requested attribute and an existing table. However, it considers these operators tools to be used and combined manually by a user in information gathering, and not as components of a larger system, such as an RDBMS.
InfoGather [10] improved the state of the art especially by identifying more candidate tables than the naïve matching approach, by introducing Web table similarity measures and identifying tables indirectly matching the query through them. In a follow-up paper [12], the system was extended to explicitly assign labels for time and units of measurements to tables, allowing for more targeted retrieval of specific attribute variants. While these two systems improve much on the accuracy of entity augmentation, they do not consider integration of this operation within more complex (SQL) queries.
Our REA system [5], which forms the basis for our DrillBeyond implementation, extends those systems by allowing top-k entity augmentation instead of single answer augmentation. This gives users a new way to deal with the ambiguity of web data-based query results by offering alternative solutions. Apart from these systems, there are many more works that employ web tables for various purposes, such as materializing tables from keyword queries [9], or answering single fact queries [11].
A second area of related work is about integration of new data sources or new types of data into traditional RDBMS. CrowdDB [6] is a very similar system to DrillBeyond in that it also enables open-world queries in a classic relational context. It employs a crowdsourcing approach to complement missing values or tuples. In contrast to that, the DrillBeyond system retrieves them in a semi-automatic fashion from structured open web tables.
The MCDB system [8] deals with Monte Carlo simulations in an RDBMS context. In this context it also deals with multiple variants of tuples, and employs a technique called "tuple bundling", in which the variants of a tuple can be bundled into one tuple when the common attributes need to be accessed, and unbundled into its variants when varying attributes to be processed. This is similar to our idea of the ω/Ω splitting of the DrillBeyond operator.
Furthermore there is a class of work on multi-query optimization, that tries to reuse intermediate results from previous queries to speed up to execution of current ones, for example [13]. This is similar to our invariant caching. Finally, the DrillBeyond system uses techniques known from the field of adaptive query processing [3][7], such as run-time reoptimization, to deal with the lack of knowledge about the data sources at plan-time.

7. CONCLUSION

In this paper, we presented the DrillBeyond system, an RDBMS / IR hybrid system, that processes SQL queries in which a user may add arbitrary attributes, that need not to be defined in the existing database. DrillBeyond processes these queries by tightly integrating a top-k entity augmentation system, that searches a large corpus of web data sources for possible ways to augment the local relations with the missing attributes. To solve the uncertainty and ambiguity of such an automated integration process, DrillBeyond answers these queries with multiple alternative query results, each based on different external data sources.
We discussed integration challenges and design choices when integrating the two types of systems, and proposed various optimization techniques to minimize the overhead of processing multi-variant SQL queries. We implemented the system in PostgreSQL and evaluated it on modified TPC-H queries, showing how our optimizations allow for efficient multi-result processing.

8. REFERENCES

[1] TPC Benchmark H. http://tpc.org/tpch, 2015.

[2] M. J. Cafarella, A. Halevy, and N. Khoussainova. Data integration for the relational web. *VLDB*, pages 1090–1101, 2009.

[3] A. Deshpande, Z. Ives, and V. Raman. Adaptive query processing. *Found. Trends databases*, 2007.

[4] J. Eberius, M. Thiele, K. Braunschweig, and W. Lehner. Drillbeyond: Enabling business analysts to explore the web of open data. *VLDB*, 2012.

[5] J. Eberius, M. Thiele, K. Braunschweig, and W. Lehner. Top-k entity augmentation using consistent set covering. *SSDBM*, 2015.

[6] M. J. Franklin, D. Kossmann, T. Kraska, S. Ramesh, and R. Xin. CrowdDB: answering queries with crowdsourcing. SIGMOD, pages 61–72, 2011.

[7] Z. G. Ives, D. Florescu, M. Friedman, A. Levy, and D. S. Weld. An adaptive query execution system for data integration. In *SIGMOD*, 1999.

[8] R. Jampani, F. Xu, M. Wu, L. L. Perez, C. Jermaine, and P. J. Haas. Mcdb: A monte carlo approach to managing uncertain data. In *SIGMOD*, 2008.

[9] R. Pimplikar and S. Sarawagi. Answering table queries on the web using column keywords. *VLDB*, pages 908–919, 2012.

[10] M. Yakout, K. Ganjam, K. Chakrabarti, and S. Chaudhuri. Infogather: entity augmentation and attribute discovery by holistic matching with web tables. In *SIGMOD*, pages 97–108, 2012.

[11] X. Yin, W. Tan, and C. Liu. Facto: a fact lookup engine based on web tables. In *WWW*, pages 507–516, 2011.

[12] M. Zhang and K. Chakrabarti. Infogather+: semantic matching and annotation of numeric and time-varying attributes in web tables. In *SIGMOD*, pages 145–156, 2013.

[13] J. Zhou, P.-A. Larson, J.-C. Freytag, and W. Lehner. Efficient exploitation of similar subexpressions for query processing. In *SIGMOD*, 2007.