No Data Left Behind: Real-Time Insights from a Complex Data Ecosystem

Manos Karpathiotakis*
EPFL
Switzerland
manos.karpathiotakis@epfl.ch

Avrilia Floratou†
Microsoft
USA
avflor@microsoft.com

Fatma Özcan
IBM Research
USA
fozcan@us.ibm.com

Anastasia Ailamaki
EPFL & RAW Labs SA
Switzerland
anastasia.ailamaki@epfl.ch

ABSTRACT

The typical enterprise data architecture consists of several actively updated data sources (e.g., NoSQL systems, data warehouses), and a central data lake such as HDFS, in which all the data is periodically loaded through ETL processes. To simplify query processing, state-of-the-art data analysis approaches solely operate on top of the local, historical data in the data lake, and ignore the fresh tail end of data that resides in the original remote sources. However, as many business operations depend on real-time analytics, this approach is no longer viable. The alternative is hand-crafting the analysis task to explicitly consider the characteristics of the various data sources and identify optimization opportunities, rendering the overall analysis non-declarative and convoluted.

Based on our experiences operating in data lake environments, we design *System-PV*, a real-time analytics system that masks the complexity of dealing with multiple data sources while offering minimal response times. System-PV extends Spark with a sophisticated data virtualization module that supports multiple applications – from SQL queries to machine learning. The module features a *location-aware compiler* that considers source complexity, and a *two-phase optimizer* that produces and refines the query plans, not only for SQL queries but for all other types of analysis as well. The experiments show that System-PV is often faster than Spark by more than an order of magnitude. In addition, the experiments show that the approach of accessing both the historical and the remote fresh data is viable, as it performs comparably to solely operating on top of the local, historical data.

*Work done while the author was at IBM.
†Work done while the author was at IBM.

SoCC '17, September 24–27, 2017, Santa Clara, CA, USA

ACM ISBN 978-1-4503-5028-0/17/09…$15.00
https://doi.org/10.1145/3127479.3131208

CCS CONCEPTS

• **Information systems** → **Parallel and distributed DBMSs**; **Data federation tools**; *Query optimization*;

KEYWORDS

Real-time analytics, ETL, data federation, distributed database systems, SQL-on-Hadoop, data virtualization

ACM Reference Format:
Manos Karpathiotakis, Avrilia Floratou, Fatma Özcan, and Anastasia Ailamaki. 2017. No Data Left Behind: Real-Time Insights from a Complex Data Ecosystem. In *Proceedings of SoCC '17, Santa Clara, CA, USA, September 24–27, 2017,* 13 pages.
https://doi.org/10.1145/3127479.3131208

1 INTRODUCTION

In the past decade, there has been an explosion in terms of data volume and variety, as well as in terms of demand for data-driven insights. Daily business operations are supported by a diverse set of applications, each with its own characteristics. Therefore, different parts of the same organization end up using different systems depending on their application requirements. NoSQL stores and OLTP systems are widely used as operational stores which store the most recent data as generated by customer transactions, user tweets, etc. ETL processes are periodically run over each operational data source to extract the data, transform it appropriately, and load it in a unifying data lake such as HDFS or a relational data warehouse, on top of which various types of analytics are performed. Querying data in such complex ecosystems is a significant challenge.

Data analysts typically use a scale-out processing system such as Spark [59] to run analytics over the data portion stored in the data lake. A major problem of accessing only the data lake is staleness, as the *tail end* of data (i.e., most recent and interesting data [1]) in the operational sources is ignored. Data staleness is often unacceptable because many applications require analysis of the tail end of the data, as well as the historical data.

To facilitate analysis over multiple data sources, engines such as Spark [59] and Hive [53] offer connectors [15, 46] to provide access to data sources that are external to the data lake. Although the connectors provide the basic mechanism to access external sources, the data analysts carry the burden of efficiently using

them. For example, a user who creates a Spark job to process both the historical data in the data lake and the most recent data in the external sources has to hand-code her analysis using low-level logic that considers the following factors: 1) the location of data as well as recent data updates in the external sources, 2) potential ETL invocations to ingest data into the lake, 3) the data overlap between the external sources and the data lake, 4) potential schema mediation between data sources, 5) optimization opportunities for the overall analysis. Going through multiple steps and writing boilerplate code before launching any type of analysis is a non-sustainable, complicated process.

Data federation systems are an established alternative for queries over multiple sources, yet they have two shortcomings which hinder their use in modern applications: First, traditional federation systems focus solely on SQL analytics. Second, they encounter difficulties optimizing queries over logical datasets that are physically spread across the data lake and an external source, and therefore exhibit suboptimal performance [4]. Thus, users end up compromising data freshness by operating only over the historical data in the data lake and ignoring the tail end of data in external sources.

This work designs a *data virtualization module* that provides a unified view over multiple data stores which are heterogeneous in terms of i) data model, ii) update rates, and iii) query capabilities. The design enables ***p**olymorphic **v**irtualization*, i.e., masking the complexity of dealing with multiple stores, while offering minimal response times [34].

To abstract away the complexity stemming from data source variety, the data virtualization module exposes a global schema on top of logically contiguous datasets which are physically partitioned across systems. The module then uses a *location-aware compiler* to map analysis from the global virtual schema to the actual sources.

The module additionally uses a *two-phase optimizer* to optimize the overall analysis and offer minimal response times. The optimizer operates in two phases to optimize both SQL and general analysis tasks, and to reduce the overall complexity of query optimization over multiple sources. Phase I considers established cost-based query optimization techniques for complex SQL queries, without being cluttered by the details of dispersed data sets. Phase II optimizes all types of data analysis by considering the properties of the underlying data sources to generate an efficient execution plan.

We validate our design by coupling the data virtualization module with the Spark framework to implement *System-**PV***. System-PV maintains all the Spark APIs and thus can support all types of Spark applications (e.g., OLAP, machine learning, etc) over a virtual, simplified schema. The location-aware compiler of System-PV rewrites analysis into a Spark script over the actual physical schema. The two-phase optimizer rewrites the resulting script using the sophisticated IBM Big SQL™ [29] query optimizer for its SQL-oriented Phase I, and the Spark SQL Catalyst optimizer [15] for its universal Phase II. As a result, System-PV efficiently serves a spectrum of choices for enterprise applications, from operating on stale data that is in the data lake, to accessing data remotely in place, as well as a combination of the two by allowing data sets to be split between the data lake (i.e., the historical part) and a remote data source (i.e., the tail end of fresh data), all while masking the actual data source and schema complexity from the users.

Overall, this paper makes the following contributions:

- We identify shortcomings of the state-of-the-art systems when deployed on top of data lake environments and accessing fresh data in external data sources (Section 3).
- Motivated by the challenges that users face, we design System-PV, a real-time analytics system that extends Spark by introducing a data virtualization module that employs a location-aware compiler and a powerful two-phase optimizer. System-PV supports and optimizes diverse analytics over a global virtual schema that masks data source variety and complexity (Sections 3-5).
- We evaluate System-PV using the TPCx-BB [28, 55] dataset appropriately extended to incorporate non-relational data, and show that System-PV is faster than Spark when accessing multiple data sources, often by more than an order of magnitude. Further, System-PV considers fresh data in external data sources at negligible performance overhead compared to operating solely on top of the data lake, while abstracting away the complexity from the user (Section 7).
- We provide insights based on our experiences operating in data lake settings (Section 8).

2 RELATED WORK

System-PV leverages decades of research in database views, ETL, and data federation systems [17, 19, 56, 57]. This section surveys these works and highlights how System-PV pushes the state-of-the-art further.

Querying Multiple Sources. In recent years, scale-out frameworks such as Spark [15], Pig [46], and Hive [53] offer specialized connectors to allow queries over multiple data sources that are "external" to HDFS (e.g., RDBMS), yet lack higher-level abstractions to hide source complexity. In addition, even when such systems perform cost-based optimizations [8], their optimizers ignore external source characteristics.

On the contrary, traditional data federation approaches have extensively studied query execution across multiple data sources [19, 30, 44, 50, 54]. However, these approaches focus solely on SQL-based data analysis and lack support for iterative or other kinds of analytics (e.g., machine learning). In addition, federated optimizers encounter difficulties when producing plans for queries that touch datasets split between multiple sources; deciding the optimal way to execute a query with multiple JOIN and UNION ALL operations over different data sources is non-trivial [4], therefore users have been avoiding such scenarios.

System-PV introduces a two-phase optimizer to specifically target cases with complex relationships between data sources, thus allowing a single logical dataset to be split between different sources, and handling data overlap. As we show later, such data distributions are frequent in data lake settings due to the periodic nature of ETL processes. Two-phase optimization was initially proposed as a way to perform site selection at runtime, and thus balance the load equally among the execution sites [18]. Then, the XPRS parallel DBMS [32] employed two-phase optimization to reduce the overall search space of possible parallel query plans. Garofalakis et al. proceeded to provide a formal framework for reasoning in terms of both single- and two-phase optimization [27]; the framework

uses metrics such as the "critical path length" of a parallel query plan, the amount of resources that an operator reserves, and the estimated execution time of an operator. Two-phase optimization can result in a final physical query plan which is different than the optimal plan [38]; still, combining a two-phase optimizer with sufficient information about the overall physical database design generally results in efficient distributed query plans [21].

Polystores. Another method to serve diverse types of queries over heterogeneous data sources is through *polystore* systems [2, 16, 22, 24, 39] that bundle together multiple query engines and use the most appropriate per query type. Polystore systems apply frequent and multi-directional data migration across the various engines [24]. Data exchange among multiple systems is challenging because it i) complicates query optimization and ii) requires connecting each system with every other system via specialized pairwise connectors [45]. The Myria [58] system uses the architecture of a federated database system as its blueprint, and operates over a polystore environment. Myria uses an extended relational, rule-based optimizer, whose rules allow expressing complex operations in ways supported by different backends. In addition, Myria uses PipeGen [31] – an underlying communication framework – to facilitate data transfer between the different backends it supports. PipeGen reduces data transfer cost by allowing data stores to exchange Apache Arrow [6] binary buffers.

Still, data transfers to and from operational data stores create additional load that can affect the stability and performance of the data stores: As opposed to polystores, we design System-PV for scenarios where the majority of data is stored in the data lake and only the tail end of the data is in external sources. In such environments, data is typically transferred from the external sources to the data lake; unidirectional communication avoids overloading the operational stores and reduces the number of plans that the optimizer considers.

ETL. ETL (Extraction, Transformation, Loading) [42, 56] is a process that populates a data warehouse with data originating in external sources. In recent years, HDFS is frequently used as the staging/destination area [48]. The popularity of HDFS has led to specialized tools [11, 13] for data ingestion. System-PV performs ETL on demand when accessing a variety of external sources, and masks ETL costs through data-source-specific optimizations.

Database Views. Database views are frequently used to mask the underlying structure of the data. System-PV supports both lazily evaluated and materialized views depending on the user requirements and the optimizer guidelines. Views are also extensively used in the domain of data integration [40], where data sources are mapped to a global schema using *local-as-view* (LAV [36]) or *global-as-view* (GAV [19]) methods. System-PV uses the GAV variation to form a global virtual schema.

3 MOTIVATION AND BACKGROUND

We now use an example to describe the challenges faced by users when developing applications that access external sources. We use Spark as a representative state-of-the-art framework [59]. Spark is frequently deployed in data lake environments because it supports various types of data analysis (e.g., OLAP, machine learning, etc.) and is compatible with various types of external sources. Spark provides both a procedural (e.g., Scala) and a declarative interface through Spark SQL [15]. Other frameworks (e.g., Hadoop [14], Hive [53], Flink [5, 10]) have similar characteristics, and their users face similar challenges.

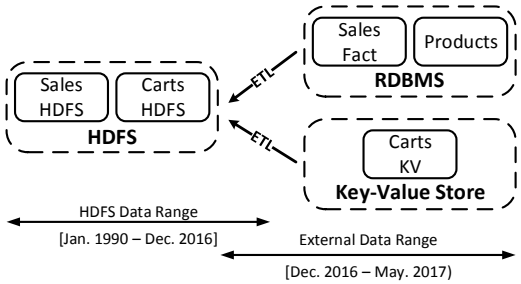


Figure 1: Typical scenario in a data lake: Analyzing recent, actively updated data along with historical data.

Motivating Example. Figure 1 depicts a modern data analysis scenario: A company uses an RDBMS to store transactional data about product sales (*Sales* dataset), and a NoSQL key-value store to store the shopping cart data of online clients (*Shopping Carts* dataset). ETL processes periodically load the data into a central data lake (HDFS), over which users run analysis using Spark. Thus, the *Shopping Carts* dataset ends up being stored across the data lake (*CartsHDFS* table) and the key-value store (*CartsKV* table). Similarly, the *Sales* dataset is spread across the data lake (*SalesHDFS* table) and the RDBMS (*SalesFact*, *Products* tables). The *Products* table is a dimension table which is frequently updated and thus remains in the RDBMS through its entire lifetime. On the contrary, the shopping cart data and the fact table of the sales data are periodically loaded to the data lake, while new data is continuously appended into their RDBMS and key-value store parts, respectively. Thus, the *tail end of the data* resides in the external sources.

```
/* HDFS side of Sales dataset */
SalesHDFS.filter("sold_date < 20161201")
/* RDBMS, normalized side of Sales */
SalesDB = SalesFact.join(Products,
      SalesFact("s_id")===Products("s_id"))
SalesDB = SalesDBAll.filter("sold_date >= 20161201")
/* Unified Sales dataset */
Sales = SalesHDFS.unionAll(SalesDB)
/* HDFS side of Carts dataset */
CartsHDFS.filter("sold_date < 20161201")
/* NoSQL side of Carts dataset */
CartsKV.filter("sold_date >= 20161201")
/* Unified Carts dataset */
Carts = CartsHDFS.unionAll(CartsKV)
/* Get number of products placed in shopping carts and
      eventually purchased */
query = Sales.join(Carts,
              Sales("user_id")===
              Carts("user_id")).count()
```

Listing 1: Spark SQL query across multiple sources.

```
query = VirtualSales.join(VirtualCarts, VirtualSales("
     user_id") ===
    VirtualCarts("user_id")).count()
```

Listing 2: System-PV query across multiple sources.

Listing 1 shows a Spark SQL query that computes the number of products which were placed in customer shopping carts and eventually purchased. The query performs a join between the *Sales* and *Shopping Carts* datasets, followed by an aggregation (Lines 16-18). Putting together the query script is non-trivial because of numerous reasons. First, a single logical dataset (*Sales* and *Shopping Carts*) consists of subsets that are physically stored across the data lake and an external data source. Thus, the user needs to be aware of the portions of these datasets that are present in each source, then manually perform the necessary filter operations to extract the correct data from each source (Lines 2, 6, 10, 12), and finally perform the appropriate union operations (Lines 8, 14). Second, these subsets might overlap. In our example, the sales data of December 2016 is stored in the data lake but is still actively updated in the RDBMS (e.g., for auditing reasons); likewise for the carts data. Thus, the user must consider her desirable query semantics in order to determine where to read the data from. In this example, the user wants to get the most recent data values, and thus she must be careful to read the data corresponding to December from the external sources instead of HDFS (Lines 2, 6, 10, 12). Third, the physical data layouts of subsets of the same dataset can differ. For example, the part of the *Sales* dataset in the RDBMS is normalized across two tables (*SalesFact*, *Products*), whereas the subset stored in the data lake is denormalized (*SalesHDFS*). Thus, the user must join the *SalesFact* and *Product* tables (Line 6). Note that this is not the case when retrieving the sales data from the data lake (Line 2). Finally, when loading the data in the lake, the ETL process might perform lightweight data transformations which must be taken into account when querying the data (not shown in this example).

As queries become more complex, the burden on the user increases; she has to hand-code more complex analysis plans, all while considering the desirable query semantics, potential data overlap, diversity in terms of data layouts, etc. In addition, every time the user wants to submit a new query, she must consider whether any of her previous assumptions have changed. Thus, query formulation over intermingled data sources becomes complex and non-declarative. On the contrary, System-PV masks source complexity by exposing a virtual schema; Listing 2 shows the System-PV query corresponding to the Spark SQL query of Listing 1. The System-PV query is significantly simpler than the Spark SQL equivalent; we will be discussing this query in detail later.

The Spark Computing Framework. We now provide a brief overview of the Spark computing framework since System-PV builds on top of it. Spark supports various types of applications (e.g., OLAP, machine learning) written as Scala, Java, and Python scripts, or as declarative queries through Spark SQL [15]. The architecture of Spark SQL is depicted in Figure 2a. Spark SQL manipulates *DataFrames*, which are distributed collections of structured records. Users express their analysis through a combination of procedural code that invokes the *DataFrame API* and declarative SQL queries that are translated to *DataFrame API* calls by Spark SQL. Regarding data access, the *Data Sources API* enables access to common HDFS formats (e.g, Avro [7], Parquet [12], etc.) and to external sources such as RDBMSs and key-value stores. Adding support for an additional data source only requires coding in a plug-in that implements the *Data Sources API*.

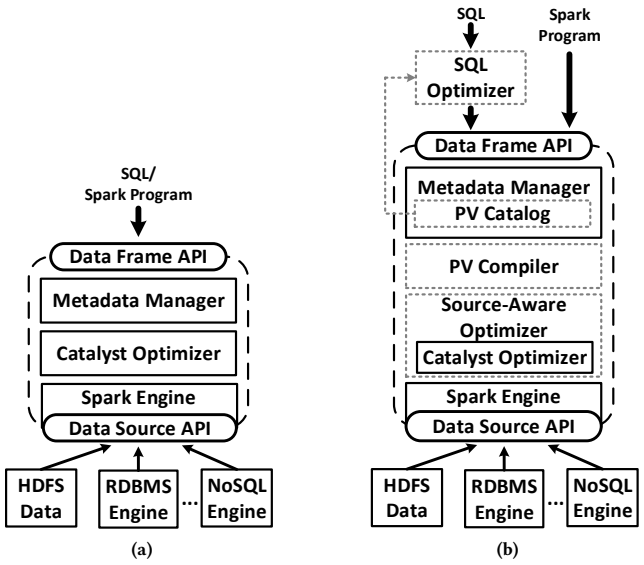


Figure 2: Architecture of (a) Spark SQL and of (b) System-PV. Dotted boxes represent extensions.

External tables and user-created *DataFrames* can be registered in the Metadata Manager (e.g, Hive Metastore [53]). Once an SQL query arrives, Spark rewrites it to the *DataFrame API* and optimizes it using the *Catalyst* optimizer. *Catalyst* currently performs logical rewrites (e.g., filter pushdown), and basic cost rewrites (e.g., choosing between a broadcast or shuffle join). Spark pushes computation to external sources when applicable. Finally, the query engine of Spark executes the resulting physical plan.

4 SYSTEM-PV

System-PV addresses the challenges related to data analysis over multiple data sources by making the following two key contributions: First, System-PV abstracts away the complexity of writing data analysis applications through a *data virtualization module* that exposes a "virtual" schema across heterogeneous data sources while still supporting all types of Spark applications. Instead of forcing the user to manually deal with data locations, ETL processes, data overlap, and conflicting schemata across sources, System-PV operates on top of *view definitions* that mask the complexity of the underlying data sources. Second, System-PV optimizes the execution of data analysis scripts using a powerful *two-phase optimizer* that supports both SQL and arbitrary analysis scripts, performs cost-based optimizations, and also considers the properties of the external sources. As a result, System-PV offers the performance of hand-coded, fine-tuned execution plans, while providing a declarative way to perform data analysis across multiple sources.

We build System PV on top of Spark, because Spark i) supports a wide range of analysis types and ii) is extensible in terms of supported data sources. System-PV serves both SQL queries as well as arbitrary data analysis scripts (typically machine learning jobs) expressed using the Spark *DataFrame API* over the global virtual schema. Figure 2b presents the high-level architecture of System-PV. The *SQL Optimizer* optimizes SQL queries. Arbitrary data analysis

scripts are passed directly to the *PV Compiler*. System-PV uses the IBM Big SQL™ [29] query optimizer to optimize the incoming SQL queries, which is based on the IBM DB2™ query optimizer, and is more sophisticated than Catalyst as it considers cost-based optimizations as well as additional query rewrite opportunities. The output of the optimizer is an optimized SQL query plan over the virtual schema, which is then expressed in the Spark *DataFrame API* and routed to the *PV Compiler*.

The *PV Compiler* rewrites the query plan in a form that references the original data sources and is understood by the Spark Engine. The PV Compiler uses the view definitions that comprise the virtual schema and are contained in the *PV Catalog*. In particular, the PV Compiler replaces each view occurrence with a sub-plan corresponding to its definition, producing an extended plan over the external data sources.

After the compilation phase, the *Source-aware Optimizer* performs a series of logical rewrites to the plan. We implement the Source-aware Optimizer as an extension of the Spark Catalyst Optimizer. Its responsibility is examining the underlying data sources and producing plans conforming to their capabilities. For example, the Source-aware Optimizer detects whether the data source targeted is an RDBMS or a NoSQL key-value store, and rewrites the logical plan accordingly. The output is a physical plan that the Spark Engine executes.

The following two sections elaborate on the System PV components: Section 5 explains how to express a virtual schema over the different data sources and launch analysis over the schema. Then, Section 6 presents the two-phase optimization process that System PV follows in order to optimize the overall analysis.

5 COMPILING CROSS-STORE QUERIES

System-PV users develop analysis scripts over a global virtual schema that abstracts away the complexity of the underlying data sources. We now discuss the properties of the virtual schema and describe how System-PV automatically rewrites user programs over the virtual schema into specialized programs that reference the external sources.

5.1 Exposing a Virtual Schema

The virtual schema consists of view definitions over datasets that are scattered across various data sources. A view provides an abstraction over a logical dataset that is physically stored in one or more data sources. We now discuss the characteristics of the view definitions.

Data Sources. System-PV supports views over both "native" and external sources. Specifically, it supports "native" Spark storage (i.e., Parquet files [12], transient in-memory *DataFrames*, and *DataFrames* cached in Tachyon [41]) and external sources such as RDBMSs and key-value stores. System-PV connects to an external source by invoking the Spark *Data Sources API*.

View Definitions. In most System-PV use cases, the view definitions that comprise the global virtual schema are *created once*; users then submit queries over the virtual schema. Note that the views need not be materialized.

To express the views, System-PV uses a subset of the relational algebra and a number of user-defined scalar functions (UDFs) that correspond to lightweight ETL primitives. The algebra which is presented in Table 1 is straightforward and allows composability of view definitions: a view can be defined based on a previously defined view. The algebraic operations take as input *views* and *expressions*. The *expressions* have different semantics depending on the operation. In the case of `Select` and `Join`, the expression filters the result, whereas in the case of `Project`, the expression projects certain columns of the dataset. `UDFunc` is an aggregating term for the various UDFs that correspond to lightweight ETL processes. Finally, a `Materializer` produces a materialized view. Depending on the value of the `mode` parameter, the view is cached as a Parquet file, a *DataFrame* stored in memory, or a *DataFrame* stored in Tachyon [41].

```
Scan(srcName)
Select(expression,view)
Project(expression,view)
Join(expression,view1,view2)
Union(view1,view2)
UDFunc(expression,view)
Materializer(expression,view,mode)
```

Table 1: Operators used in view definitions

Listing 3 shows the view definitions for our running example, which are created once. Using the view definitions, the users operate directly on the virtual schema (*VirtualCarts, VirtualSales)* and thus can be unaware of the actual data locations. Listing 2 shows the simplified System-PV query over the virtual schema that corresponds to the Spark SQL query of Listing 1.

```
1 cKVSel = Select('t >= 20161201', Scan(CartsKV))
2 cHDFSSel = Select('t < 20161201', Scan(CartsHDFS))
3 VirtualCarts = Union(cKVSel, cHDFSSel)
4 SalesDB = SalesFact.join(Products,
5     SalesFact("s_id")===Products("s_id"))
6 sDBSel = Select('t >= 20161201', Scan(SalesDB))
7 sHDFSSel = Select('t < 20161201', Scan(SalesHDFS))
8 VirtualSales = Union(sDBSel, sHDFSSel)
```

Listing 3: Views for running example, created once.

Managing Views. System-PV contains a catalog service, namely *PV Catalog*, to maintain the virtual schema. Apart from storing the view definitions, the PV Catalog captures information about each data source, such as its type and capabilities (e.g., whether the data source exposes an index or whether it supports range queries).

Whenever an ETL process loads new data in the data lake, System-PV updates automatically the view definitions in the PV Catalog. For this purpose, System-PV assigns a "watermark" to the views that capture a certain temporal range (shown in blue in Listing 3). Additionally, System-PV assigns a temporal range to each data batch loaded from the external sources to the data lake; these data batches are stored as separate HDFS partitions [37]. The range of a data batch corresponds to the period from the transaction time[1] of the oldest batch entry to that of the newest batch entry. In the example of Figure 1, loading the tail end of data into the data lake would result in a batch with the range `[Dec.2016 - May 2017)`. System-PV supports external sources that handle transactional workloads such as RDBMSs or key-value stores. If the last batch ingested into the data lake corresponds to the range $[t_1, t_2)$, then

[1]The time when the fact is (logically) current in the database [51].

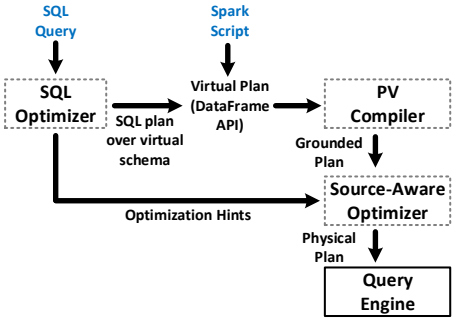


Figure 3: System-PV Pipeline.

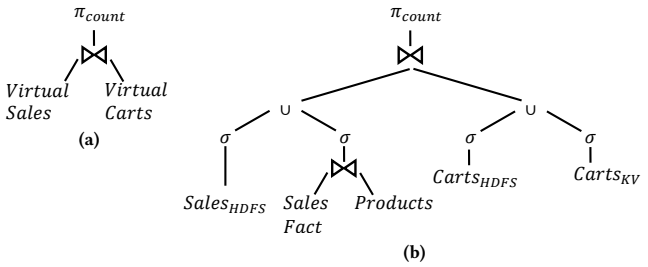


Figure 4: Virtual plan of our running example (a), and its corresponding grounded plan (b).

System-PV automatically assigns the range $[t_2, +\infty)$ to the data in the external source. When an ETL process loads a data batch, it edits the watermarks of the affected views to incorporate the temporal range of the incoming batch, thus triggering System-PV to update the view definitions.

Data Overlap. A common scenario is to have a large portion of a dataset stored in the data lake whereas the tail end of the data is stored in an actively updated external source. Depending on the nature of the application and the periodic ETL processes, it is possible that these two subsets overlap. In the example of Section 3, the sales data corresponding to the period between 1990 and 2016 is archived in the data lake (HDFS). The data for December 2016, however, is also stored in the company's operational data store because updates still occur over this data. This data will eventually be pushed to the data lake and the stale HDFS counterpart will be refreshed. Until then, System-PV enables users to define a view that specifies which side (HDFS or the RDBMS) should serve the overlapping data. This view is defined based on the application requirements: If data freshness is important, then the data corresponding to December 2016 must be fetched from the RDBMS as shown in Line 6 of Listing 3. Otherwise, accessing the local HDFS data is prone to be more efficient.

Lightweight ETL. System-PV handles datasets that are physically split across the data lake and an external source even when the corresponding data subsets have different schemata. Specifically, System-PV offers primitives for expressing lightweight ETL processes. Users can remap schemata by changing the name, datatype, and the order of fields. In addition, users can employ UDFs that transform column values in order to, for example, convert different units of measurement and handle out-of-bound values (e.g., negative ages). System-PV also handles more complex cases such as the one presented in Figure 1, where the *Sales* data is normalized in the RDBMS but it is denormalized in the data lake. As shown in Line 4 of Listing 3, users can express views using join operations to denormalize the external data at query time.

5.2 Querying over a Virtual Schema

As shown in the example of Listing 2, System-PV users express their scripts directly over a virtual schema. At some point, System-PV must therefore translate the virtual schema to the actual heterogeneous data sources. Figure 3 shows how PV Compiler performs the translation: When a user expresses an SQL query or a procedural script, System-PV generates a logical plan over the virtual schema that is described using the *DataFrame API*; we call this a *virtual plan*. Then, System-PV feeds the *virtual plan* to the *PV Compiler*, which in turn uses the view definitions stored in the PV Catalog to rewrite the plan into a *grounded plan*; the grounded plan references the original data sources and is understandable by the Spark engine. The *virtual* and *grounded* plan corresponding to our running example are depicted in Figure 4.

Specifically, the PV Compiler traverses the *virtual plan* until it locates scan operations corresponding to virtual datasets. For each of the virtual datasets, the PV Compiler looks up its view definition in the PV Catalog, and outputs code that describes how to access the corresponding data in the external data sources. The PV Compiler performs the rewriting using two components: an *Algebraic Rewriter* and an *Expression Rewriter*. The Algebraic Rewriter takes as input a view definition, maps the operators of the view into equivalent operations of the *DataFrame API*, and calls the Expression Rewriter to transform expressions when necessary. For a view defined as `Select('x < 10',Scan(table))`, the Algebraic Rewriter invokes the Spark SQL `filter()` function, and the Expression Rewriter produces the code for the predicate evaluation.

Most of the algebraic nodes of Table 1 have 1-1 mappings to Spark operations, similar to the ones of `Select`. System-PV models `UDFunc` operations as overloaded versions of a projection operation. Finally, the `Materializer` operator is mapped to a different type of Spark operation (e.g., a persistent flush command, transient in-memory caching, etc.) based on a `mode` parameter specified at view definition time.

Summary. System-PV masks data source complexity by exposing a global virtual schema and by using a location-aware compiler to generate specialized scripts that access the external sources. System-PV also alters its view definitions to cater for ETL-triggered data updates.

6 A TWO-PHASE OPTIMIZER FOR CROSS-STORE ANALYTICS

System-PV allows users to perform their analysis over a global virtual schema, thus masking source complexity. Nevertheless, by enabling data analysis over a wide combination of heterogeneous data sources, the query optimization task seemingly becomes harder. Distributed query optimization is a well-studied problem [38], which is intensified in our case because i) System-PV supports different

types of analysis besides relational queries and ii) heterogeneous data sources (e.g, NoSQL stores, RDBMSs) are accessed in the same analysis task.

System-PV makes use of a sophisticated two-phase optimizer. As shown in Figure 3, System-PV applies the first optimization phase to SQL queries only. System-PV applies the second phase regardless of whether the data analysis is expressed using SQL or an arbitrary Spark program. Specifically, when receiving an SQL query, System-PV applies the cost-based optimizations of a mature SQL optimizer by considering **only** the virtual schema (***Phase I***). System-PV further optimizes the analysis plan by exploiting the capabilities of the underlying data sources (***Phase II***).

The IBM Big SQL optimizer performs numerous cost-based optimizations over an input SQL query, yet is unable to reason in terms of non-relational types of analysis such as Spark SQL procedural scripts. On the contrary, the Spark SQL Catalyst optimizer can process any type of analysis expressed in the Data Frame API – relational or not. Therefore, Phase I uses the specialized Big SQL optimizer so that it specifically target SQL analysis, and Phase II uses the Catalyst optimizer so that it is compatible and applicable to any type of Spark SQL analysis.

System-PV keeps the two optimization phases separate for two reasons: First, compared to optimizing procedural data scripts, optimizing declarative SQL queries is a more nuanced process, requires examining multiple execution plans, and typically benefits more from complex query optimization. Therefore, System-PV applies Phase I over SQL queries and not arbitrary data scripts. Second, the separation confines the universe of decisions in each phase. Unifying the two phases complicates plan enumeration: The source-specific rewrites of Phase II expand the query plan and thus increase the optimization space, so exposing the complexity to the SQL query optimizer would complicate its major task of identifying the appropriate join order.

6.1 Phase I: SQL Optimization

Optimizing SQL queries in a distributed setting is a challenging, error-prone task [3, 23, 25, 26, 30, 38, 43]. In the case of Spark, the Catalyst optimizer is a promising first step, but at the time of writing, it mainly focuses on simple rewrites, and it supports very few cost-based optimizations. System-PV therefore uses the IBM Big SQL federated query optimizer because it supports sophisticated rewrites and cost-based optimizations.

As depicted in Figure 3, when System-PV receives an SQL query over the virtual schema, it routes the query to the SQL Optimizer. The SQL Optimizer requires data source information to perform costing and to come up with an efficient query plan; System-PV thus exposes such information for every "virtual table", based on the metadata and statistics stored in the PV Catalog.

Specifically, when a dataset is split across an external source and the data lake, System-PV distinguishes between two cases. When the ETL process that loads the data in the data lake is frequent ("frequency" is measured based on a tunable threshold), the SQL optimizer considers only the data in the data lake. The tail end of data is thus masked during optimization Phase I and is considered only in Phase II. System-PV masks the tail end during Phase I for the following reasons: i) the tail end of data is typically small compared to the overall dataset, which is the default case in data lake environments, and ii) exposing more complex view definitions that capture the full dataset can complicate plan enumeration [4]. On the other hand, when ETL is sporadic, the SQL optimizer considers only the remote data source, since the remote data access dominates query execution costs.

System-PV exposes the local HDFS cluster to Big SQL as the "primary" data source, and the rest of the data sources as remote data stores. Depending on the data source exposed, the optimizer identifies the source capabilities (e.g., ability to perform projection pushdown, indexes) through specialized *source wrappers*[2] [17, 44]. Each source wrapper exposes data statistics to Big SQL in order for it to compute the overall query cost. Big SQL offers sophisticated, statistics-aware wrappers for RDBMS. On the other hand, Big SQL lacks a source wrapper for distributed key-value stores such as Cassandra [9]. System-PV therefore emulates the connection with an instance of Cassandra by re-using an existing wrapper: Specifically, given that Cassandra is a distributed key-value store, System-PV uses a wrapper designed for a parallel RDBMS, and informs Big SQL about a hypothetical hash index over the mock RDBMS in order to emulate Cassandra's key-based accesses. In addition, System-PV specifies a data partitioning scheme that the mock RDBMS hypothetically uses (e.g., hash partitioning) in order to emulate the partitioning scheme employed by Cassandra [20]. Finally, System PV collects statistics over Cassandra and injects them in the PV Catalog. Overall, System PV uses the different source wrappers of Big SQL and the accumulated data statistics to make well-informed decisions for SQL query optimization.

The SQL Optimizer uses the information of the exposed data sources to produce an optimized logical query plan over the *virtual schema*. In addition, it produces information about the corresponding physical plan. For example, the optimizer indicates the physical join algorithms to be used, and potential intermediate result materializations. System-PV uses the information about the physical plan as optimization hints during the source-aware optimization that produces the final, physical plan (Phase II).

The optimized logical query plan is forwarded to the PV Compiler, which rewrites any occurrences of views and generates the *grounded plan* that references the original data sources. The *grounded plan* along with the optimization hints are then passed to the Source-aware optimizer used in Phase II.

6.2 Phase II: Source-aware Optimization

The second optimization phase applies source-specific optimizations to all data processing tasks, regardless of whether they are expressed in SQL or procedural code, through use of the *Source-aware Optimizer*.

An issue of Catalyst is that it misses multiple optimization opportunities for queries over external sources. Specifically, Catalyst uses the *Data Source API* to access external sources. The *Data Source API*, however, is meant for single-table accesses. As a result, only selections and projections are pushed down to the external sources.

[2]The data source wrappers of Big SQL are not to be confused with the data source connectors of the Spark Data Source API; the former are used during query optimization, whereas the latter only perform data access during query execution.

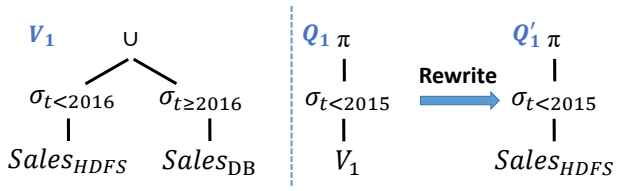


Figure 5: Query plan simplification during source-aware optimization.

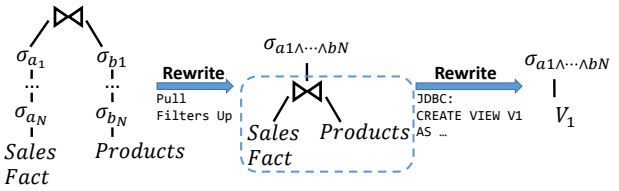


Figure 6: Join pushdown rewriting during source-aware optimization.

More complex operators such as joins are not pushed down, thus often missing opportunities for reducing the network traffic. Even worse, when the underlying external source is not an RDBMS, few query templates allow pushdown of any operator.

As shown in Figure 3, the input of the *Source-aware Optimizer* comprises i) a *grounded plan* that references the original data sources, and ii) the optimization hints produced by the SQL optimizer. The Source-aware Optimizer extends Catalyst with different categories of rewrite rules. The first category simplifies the *grounded* plan and applies the optimization hints to improve the physical plan quality. The second category maximizes operator pushdown. Finally, the third category examines each data source type in isolation and applies targeted optimizations. We now elaborate on each category.

Rewriting Internal Plan Nodes. After the PV Compiler expands the definitions of views, the resulting *grounded plan* becomes more complex because additional operations such as unions and selection predicates are exposed. The Source-aware Optimizer simplifies the plan by pruning redundant sub-trees and coalescing filtering expressions into disjunctive normal form.

Figure 5 presents an optimization instance over the rolling example of Figure 1: V_1 is a view that models a union between HDFS and RDBMS-resident data of the *Sales* dataset. Both sides of the union have a filtering predicate applied. When the Source-aware Optimizer examines the filtering predicate of Query $Q1$, it detects that the *Sales* data in the RDBMS does not need to be accessed to answer the query. It thus rewrites the plan to access only the HDFS-resident data.

After simplifying the plan, if the original analysis task was an SQL query, the Source-aware Optimizer enforces the optimization hints suggested by the SQL Optimizer during Phase I. Specifically, if the SQL optimizer suggests that a join operation must broadcast the smaller dataset involved, the Source-aware Optimizer rewrites the plan accordingly to use the appropriate Spark broadcast hash-join operation. The SQL optimizer may also suggest that a sub-tree of the overall query plan must be materialized and then reused later in the same query. In this case, the Source-aware Optimizer injects a *Materialize* operator in the physical query plan.

Operator Pushdown. When dealing with remote data sources, it is important to reduce the amount of data movement through the network by pushing down operations to them. The *Data Source API* enables some basic selection and projection pushdown for queries over external sources. However, Catalyst has two major limitations: First, Catalyst is often unable to push down more complex filtering predicates. Second, Catalyst is unable to push down more complex operators such as joins, because the *Data Source API* of Spark SQL is restricted to single-table data accesses. System-PV must thus adress both limitations in a non-intrusive manner, so that it remains compatible with "vanilla" Spark SQL.

First, the Source-aware Optimizer further simplifies filtering predicates in order to push them down to the external data sources. Second, System-PV performs join pushdown by adding an optimization pass that proceeds as follows: The pass traverses the query plan and finds the largest subtree that contains data accesses to a single data source. If System-PV detects such a subtree, it makes a call to the underlying source to define a temporary view representing the subtree. By exposing the subtree as a single table, System-PV supports join pushdown without harming compatibility with the *Data Sources API* of Spark SQL. Figure 6 presents an application of the join pushdown rule over the example of Figure 1: Initially, any selection predicates are *pulled above* the join operation, so that the optimization pass has simpler tree patterns to detect. Once a join pattern between two original relations is detected, a temporary view V_1 is created. Finally, selection pushdown is re-applied on the final view, which can be deleted once the query terminates.

Exploiting Source Characteristics. Unlike vanilla Spark, System-PV takes into consideration the characteristics of the different underlying data sources in order to further optimize the analysis plan. Specifically, the Source-aware Optimizer rewrites queries that are submitted to external data sources in a way that masks the data movement costs.

Large-scale applications pay a significant cost to serialize data, transfer it over the network, and deserialize it [33, 47]. The cost is even more pronounced for Spark when it accesses external data sources: In case of RDBMSs, Spark blindly submits each query through a single JDBC connection; a single Spark task executor is responsible for receiving the data through the network, deserializing it, and shipping results to the other executors to continue query execution. This single task executor often becomes the bottleneck. System-PV, on the other hand, masks data movement cost by rewriting the query into a semantically equivalent union of multiple queries that are concurrently submitted to the RDBMS by multiple Spark task executors. Specifically, the Source-aware Optimizer applies an optimization pass that splits an RDBMS scan operation into a union of scan operations. The optimization pass is triggered when the data to be scanned i) has an index, or ii) is range-partitioned on the query's predicate(s), which is typical in modern deployments [37]. In these cases, the RDBMS performs selective data accesses which further improve execution times.

System-PV performs a similar optimization when accessing key-value stores, which by design are optimized for queries requesting a single data item by key. The Source-aware Optimizer rewrites range queries on the key attribute into a union of equi-predicate selections

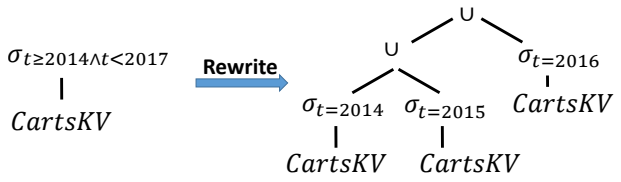


Figure 7: Range query rewriting during source-aware optimization: Data accesses become parallelizable.

to parallelize the ingestion on the Spark side, and also better suit the query capabilities of the key-value store. Figure 7 presents an application of said optimization over the example of Figure 1: The range predicate is split into a number of equi-predicate selections, and the results of the sub-queries are unified.

Summary. System-PV uses a two-phase optimizer to cover both SQL queries and general analysis tasks, and to reduce the complexity of optimization over multiple data sources. In Phase I, an SQL Optimizer applies SQL-centric optimizations. In Phase II, the Source-aware Optimizer considers the properties of the underlying data sources.

7 EXPERIMENTAL EVALUATION

We experimentally evaluate System-PV by emulating a business intelligence scenario similar to that of Figure 1. The majority of the data is in the data lake (HDFS), whereas the tail end of the data is in a data warehouse (IBM DB2® DPF™) and a key-value store (Cassandra [9]). Our key results are the following:

(1) System-PV is faster than Spark – often by more than an order of magnitude – while masking the complexity of accessing multiple data sources (Section 7.2) .

(2) The SQL Optimizer of System-PV produces better query plans than Catalyst (Section 7.2.1).

(3) The Source-aware Optimizer of System-PV provides significant performance gains by masking the data transfer costs through better parallelization (Section 7.2.2).

(4) System-PV accesses the remote data tail end with small overhead added to the case of operating solely on top of the historical data in the data lake (Section 7.3).

7.1 Experimental Setup

We use the TPCx-BB benchmark [28, 55] data generator at scale factor 1000 to populate the **web_clickstreams** table (180 GB) and **web_sales** table (450 GB). To incorporate non-relational data, we additionally generate the **web_events** dataset (90 GB) that contains sales data that has been produced by mobile devices in JSON format. The **web_clickstreams** table is entirely stored in the data lake to emulate the case in which data is directly ingested in HDFS. The **web_sales** table is split between HDFS and DB2 DPF. This is because information about sales is typically inserted in an RDBMS and periodically loaded in the data lake. Similarly, the semi-structured **web_events** dataset is split between HDFS and Cassandra.

We use Spark version 1.4.0 on a 10 node cluster, DB2 DPF version 10.1.0 on a 5 node cluster, and Cassandra version 2.1.7 on a 4 node cluster. All nodes are equipped with two 6-core Intel Xeon E5-2430 CPU @ 2.20GHz, 96GB RAM, and 11 × 2TB SATA disks. The nodes are connected through a 10 Gbit Ethernet switch.

The experiments compare four *data placement configurations*: In the first case, 90% of the Sales and Events tables reside in HDFS, and the 10% left resides in DB2 and Cassandra, respectively (**90-10**). In the second case – the closest to real-world scenarios – 99% of the Sales and Events tables reside in HDFS, and the 1% left resides in DB2 and Cassandra, respectively (**99-1**). In both cases, data is range-partitioned based on a date attribute. Finally, the third and fourth cases represent baseline extremes: Either all datasets are entirely stored in HDFS (**Local**), or each dataset resides in a different data store (**Remote**). **Local** represents the scenario where the users access only the data in the data lake and thus ignore data freshness.

We use a query template that represents a scenario which is frequent in data lake environments: combining data from all the involved data sources. The template $T(X, Y, Z)$ – shown below – includes a 3-way join and a number of filtering predicates with non-fixed selectivities (X, Y, Z). The template allows us to generate various types of queries that stress different parts of a system. By using different combinations of predicate selectivities, we affect the amount of data to be transferred across sources, and also evaluate the query processing and optimization capabilities of System-PV, given that different selectivities can trigger different join orders.

```
SELECT AVG(s_sales_price)
FROM web_clickstreams c
JOIN web_sales ss ON
     (c_user_sk = s_bill_customer_sk)
JOIN web_events e ON
     (s_bill_customer_sk = e_cust_id)
WHERE (c_click_date_sk BETWEEN X1 AND X2)
 AND  (s_sold_date_sk BETWEEN Y1 AND Y2)
 AND  (e_session_date BETWEEN Z1 AND Z2)
```

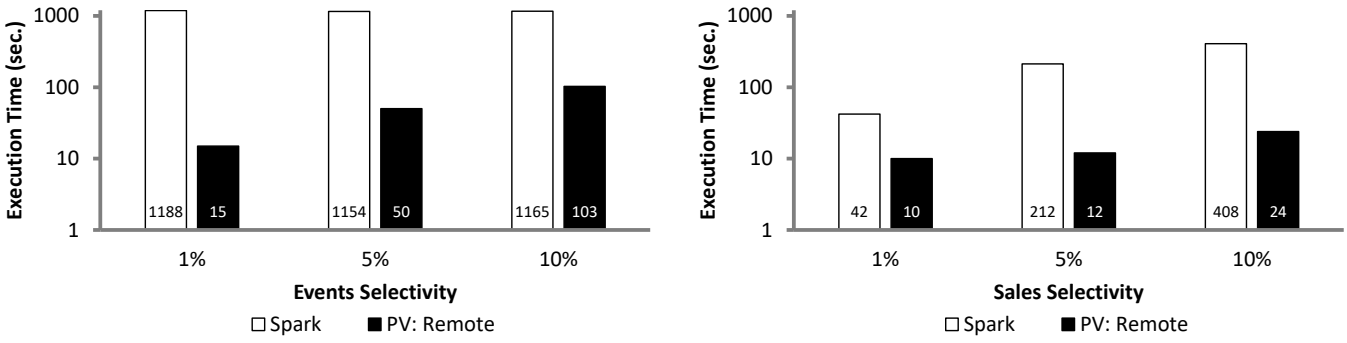
Listing 4: Query template for analysis across data sources.

7.2 System-PV vs. Spark

We compare System-PV with Spark by quantifying the impact of each of the two System-PV optimization phases.

7.2.1 System-PV SQL Optimizer vs. Spark Catalyst. The goal of this experiment is to validate that the SQL optimizer of System-PV produces efficient plans. We generate the instantiation $T(1, 5, 10)$ of the query template in Listing 4, namely Q, which selects 1% of the Clickstreams data, 5% of the Sales data, and 10% of the Events data using the **90-10** data placement configuration. We compare the query plan generated by the System-PV SQL optimizer for Q (Query Plan No.1 in Figure 9) against various other plans in the space of all plans for queries generated from the template of Listing 4.

We choose not to pick random plans from the plan space for this comparison because they are highly likely to exhibit dramatically poor performance. Instead, we pick plans that are potentially close to the optimal. One such plan is the one generated by the Catalyst optimizer (Query Plan No.9). Other selected plans were the ones generated by the System-PV SQL optimizer for various other template instantiations. These plans are shown in gray in Figure 9. We execute multiple runs of query Q **on System-PV**, each time hand-coding a different virtual plan corresponding to one of the selected plans, and using the same source-aware optimizations for all of them.



(a) Clicks and Sales Selectivities kept at 0.1%.

(b) Clicks & Events Selectivities kept at 0.1%.

Figure 8: Spark is unable to keep up with System-PV even for very selective queries.

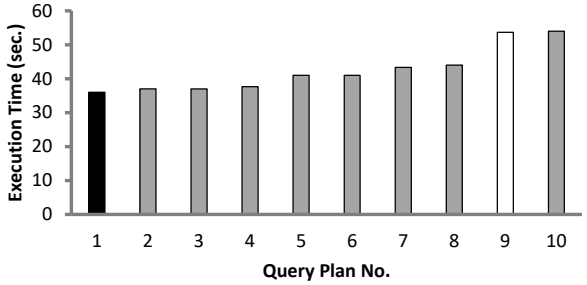


Figure 9: The SQL Optimizer of System-PV picks the best candidate plan (No. 1), whereas Spark's Catalyst optimizer picks plan No. 9.

Different plans lead to different execution times – a fact that further highlights the need for a cost-based optimizer. System-PV picks plan No.1 (*(Sales ⋈ Events)* ⋈ *Clicks*), which builds hash tables on the Clickstreams and Events datasets (i.e., the right operands of each join) and probes them using records from the Sales dataset (i.e., the left operand). This plan ends up being the best choice because the Clickstreams dataset is stored in the data lake and thus System PV builds a hash table over each node's local data in parallel. The Catalyst optimizer, on the other hand, picks plan No.9 (*(Clicks ⋈ Sales)* ⋈ *Events*) – the second worst from the plans tested. Plan No.9 builds hashtables over the Sales and the Events datasets, which it then probes using the records of the Clickstreams dataset. Both Sales and Events, however, have a significant portion of data stored in remote sources and thus require additional effort to build the hashtables. In addition, their corresponding predicates are less selective than the predicate on the Clickstreams dataset.

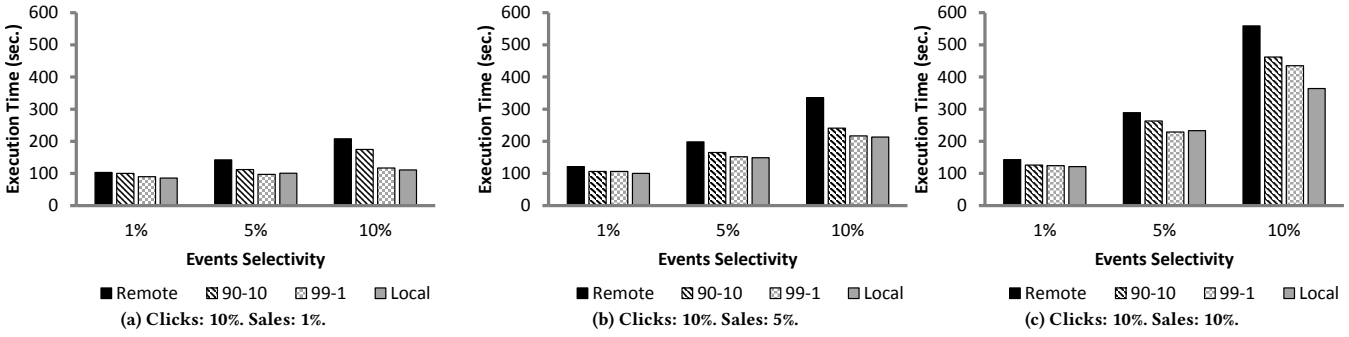
Our results show that unlike Catalyst, the System-PV SQL optimizer considers the predicate selectivities and the location of a dataset over which a hashtable is built to produce an efficient plan. We repeated this analysis using the same protocol but starting with different instantiations of the template T using other selectivity values and data placement configurations as well: We obtained similar results which we omit for brevity.

7.2.2 Impact of Source-aware Optimizer. We now quantify the performance gains that System-PV has over Spark due to the Source-aware Optimizer. We make sure that both System-PV and Spark use the same optimal *virtual plan* by hand-coding the plan produced by the System-PV SQL optimizer. As shown in Section 7.2.1, in many cases Spark picks a suboptimal plan, and thus the Spark performance results presented here are conservatively optimistic.

We test the **Remote** data placement configuration – the most challenging of the ones examined – by instantiating the template T with different selectivity values for the predicates; we generate 6 queries in total. The predicates touching two of the three datasets are kept very selective. Less selective configurations stressed Spark even more; we omit them in the interest of space. We vary the selectivity of the predicate over the third dataset, so that the amount of data fetched from the remote source varies too.

Figure 8a presents the case in which only 0.1% of the HDFS-resident clickstreams and the DB2-resident Sales are selected. The selection predicate for the Cassandra-resident Events ranges from 1% to 10%. In this case, System-PV is 11× to 79× faster than Spark. Note that the execution time of System-PV increases as the query becomes less selective, and more data has to be fetched from Cassandra. Spark, on the other hand, shows little variation in execution time regardless of the amount of data to be fetched. The reason is that Spark attempts to push a range (sub-)query down to Cassandra, which Cassandra is unable to process. Thus, Cassandra ships the entire dataset to Spark through a single-threaded connection, and Spark then applies locally the range predicate. On the contrary, System-PV rewrites the range query into a union of equi-predicate selections that it concurrently submits to Cassandra. These standalone sub-queries are served in parallel, thus resulting in fast data ingestion rates.

Figure 8b presents the case in which 0.1% of the HDFS-resident Clickstreams and the Cassandra-resident Events are selected. The selection predicate for the DB2-resident Sales ranges from 1% to 10%. Note that for this experiment, we incorporated the System-PV optimizations targeted for key-value stores into Spark SQL. These optimizations were enabled in both Spark SQL and System-PV in order to quantify the performance benefits attributed to the database-related rewrites of System-PV in isolation. System-PV is again 4× to 17× faster than Spark because it parallelizes



(a) Clicks: 10%. Sales: 1%. (b) Clicks: 10%. Sales: 5%. (c) Clicks: 10%. Sales: 10%.

Figure 10: System-PV performance for various data placement configurations and query selectivities.

data transfer from DB2. Note that Spark SQL *does* successfully pushdown the selection predicate to DB2, but retrieves the data through a single-threaded connection, which ends up being the bottleneck for the entire query.

7.3 System-PV Performance

We now evaluate System-PV using all four data placement configurations and varying the amount of fresh data transferred over the network. Our aim is to verify that System-PV performance for split data scenarios is comparable to the scenario of solely operating on top of the historical, stale data. We exclude Spark from this discussion because its running time always exceeded 1000 seconds; as Section 7.2.2 showed, Spark is unable to keep up with System-PV even for selective queries that require small network data transfers. The results show that System-PV uses the optimizations of Section 6.2 to mask the cost of remote data accesses, and thus provides similar performance to solely operating on top of the data lake.

Figure 10 presents 9 instantiations of the query template. All of them select 10% of the HDFS-resident Clickstreams. The queries select either 1% of the Sales dataset (Figure 10a), 5% (Figure 10b), or 10% (Figure 10c). We vary selectivity over the Events dataset in every query to gauge the effect of accessing the slowest data source (i.e., Cassandra).

As seen in Figure 10a, all the data placement configurations have similar performance, with **Remote** being slightly slower than the others. The performance gap opens in the case of 10% selectivity; even then, however, the performance observed with the **99-1** configuration is almost identical to the best-case scenario where remote data access does not occur (**Local**). The reason is that the source-aware optimizations mask the remote data access cost by overlapping data transfer with query execution.

The queries shown in Figure 10b are more expensive than the ones of Figure 10a because a bigger subset of the Sales dataset participates in the join. Still, the **90-10** and **99-1** configurations exhibit execution times similar to the **Local** configuration. Even in the case of 10% selectivity, the execution times corresponding to the **99-1** and **Local** configurations are only 4 seconds apart, thus denoting that System-PV is again able to mask remote data accesses.

Figure 10c presents the least selective version of the experiment. When selectivity over the Events dataset is 1%, all data placement configurations except **Remote** have almost identical execution time. Note that although the sub-query pushed down to DB2 is non-selective, System-PV splits and parallelizes the sub-query, thus hiding the increased data transfer cost. When selectivity over the Events dataset reaches 10%, the gap between **Remote** and **Local** increases. Note, however, that the **Local** configuration misses the latest fresh data. The performance difference stems from the simultaneous increase of i) remote data accesses, ii) the amount of data shuffled due to the distributed hash join, and iii) the size of intermediate results, all of which stress the network bandwidth. Finally, the performance of System-PV in the common split-dataset configurations (**99-1**, **90-10**) is similar to that observed when accessing all the data locally (**Local**).

Summary. System-PV significantly outperforms Spark, even for the worst-performant scenario of accessing federated data sources (**Remote**). In addition, when testing System-PV under different data placement configurations, the response times for the two extreme cases (fully local vs. fully remote) start to diverge; still, for the split-dataset cases that System-PV targets, response times are comparable to that of the best-performant, fully local scenario (**Local**), without compromising data freshness.

8 PERSPECTIVES

Our experience with Spark and other similar frameworks has shown us that although they support various types of data analysis over historical data in a data lake, they lack the necessary abstractions to query data sets spread across multiple data sources, thus rendering the overall analysis complex for the user. At the same time, their performance is suboptimal when accessing external sources.

System-PV introduces a high-level abstraction in the form of a global virtual schema, which hides source complexity from users and allows them to seamlessly access both the historical as well as the latest data. System-PV also optimizes both SQL and procedural analysis tasks through a unique two-phase query optimization approach. System-PV thus supports a broad spectrum of data usage patterns: an individual dataset can be accessed in the remote source completely, can be split between the data lake and the remote source, or can be accessed locally in the lake (if the application can tolerate data staleness). In addition, splits of a dataset can overlap; System-PV chooses from which source to retrieve the overlapping part depending on the user's data freshness requirements.

Using System-PV in practice has led us to a number of observations that allow reaching its full potential, and that can be useful as guidelines to system designers working on split-data scenarios.

8.1 System-PV for enterprise workloads

Enteprise data management architectures typically model data using a variation of the star or the snowflake schema, which involve few large fact tables and numerous smaller dimension tables [35].

Small datasets. System-PV masks the cost of accessing remote datasets of small size, such as dimension tables of a star schema. Given that dimension tables receive frequent updates, and that different parts of an organization often join their own versions of dimension tables against a fact table [35], we propose storing dimension tables only in the original, external data sources; there is no need to store them in the data lake as well, since accessing them with System-PV has minimal overhead.

Fact tables. Large fact tables receive append-like updates, and users typically set up an ETL process to archive the data appends in the data lake. System-PV by default accesses both the local and the remote part of a fact table. If possible, we suggest running ETL frequently, so that running analysis with System-PV over both parts of the fact table has comparable performance to accessing only the local part. In addition, more data accumulates in the lake over time, whereas the size of the remote delta remains stable, therefore the cost of remote accesses appears small due to the order-of-magnitude difference in local and remote data sizes.

Minimizing data transfers. The source-aware optimizations of System-PV that generate sub-queries to parallelize external data retrieval provide their maximum benefit when the external sources offer a way to reduce the amount of data that each sub-query accesses. For key-value stores, a query on the key of each object naturally accesses a small amount of data. For RDBMS, populating indexes on fields that are popular query predicates, or partitioning the data, is helpful. Given that primary and foreign keys are typically coupled with indexes, enterprise star and snowflake schemata already have useful indexes in place. Therefore, System-PV applies its rewriting optimizations without requiring an additional indexing storage overhead.

8.2 Optimizing SQL-on-Hadoop performance over multiple sources

Apart from the user-friendly virtual schema that System-PV employs, it also makes use of multiple performance optimizations that improve the performance of Spark scripts over dispersed datasets. It is worth examining whether these optimizations can also be applied to existing systems even if said systems currently lack first-class support for data virtualization. There is a number of ways in which existing SQL-on-Hadoop systems can be adjusted to improve their performance over diverse data stores. We use Spark SQL as an example, and consider its architecture in a top-down fashion.

Starting from the query optimizer, Catalyst is a significant effort towards performing optimization across multiple types of analysis. However, it is currently not as mature as several traditional, specialized database optimizers that have been refined over multiple years. Thus, we believe that Catalyst must also introduce interfaces that allow users to "plug" their optimizer of choice based on the type of analysis they intend to launch[3]. Users can choose among optimizer modules such as the one of System-PV, Orca [52], Calcite [8], etc.

Our experience building System-PV showed us that integrating the Source Optimizer's rewrite rules into Catalyst is straightforward and would be a valuable addition to Spark. Still, applying the source-aware rewrites of System-PV requires examining carefully the properties of the underlying systems, and triggering the rewrites judiciously. For example, triggering the query rewrite for range predicates that access non-key fields in a key-value store, or for arbitrary, non-partitioning / non-indexed fields in a DBMS table, can significantly penalize performance. Therefore, Spark must be able to acquire and store information/statistics from the underlying data stores in order to make educated rewriting decisions.

Instead of applying some of the source-aware optimizations in Catalyst, one could extend/rewrite the data connectors of Spark to reduce the cost of accessing and transferring remote data into the data lake. As shown by this work, one way to reduce the cost is by parallelizing the sub-query that accesses a remote store. In addition, data connectors can perform data exchange using a portable, binary wire format such as Arrow [6]; Arrow has the same in-memory and on-wire representation, and thus reduces the effort spent in data (de)serialization, which is a major cost in data-center-scale analytics [33, 47].

Summary. System-PV provides a spectrum of choices for data freshness and where to access the data in complex enterprise data ecosystems. Combined with the guidelines above, System-PV forms a comprehensive solution for ad-hoc data analysis in enterprise settings, which can also influence the design of state-of-the-art SQL-on-Hadoop systems.

9 CONCLUSIONS

We present System-PV, a system that supports various types of analysis over multiple data sources. System-PV addresses the shortcomings of state-of-the-art systems by extending Spark with a data virtualization module that masks data source complexity. System-PV uses a location-aware compiler and a sophisticated two-phase optimizer to optimize user scripts over a global virtual schema. Our results show that System-PV is significantly faster than Spark when querying dispersed datasets, and introduces small overhead for accessing the remote tail end of the data compared to operating solely on top of the data lake.

Acknowledgments. We would like to thank the reviewers for their valuable comments and suggestions on how to improve the paper. This work is partially funded by the EU FP7 programme (ERC-2013-CoG), Grant No 617508 (ViDa).

REFERENCES

[1] Cristina L. Abad, Nathan Roberts, Yi Lu, and Roy H. Campbell. 2012. A storage-centric analysis of MapReduce workloads: File popularity, temporal locality and arrival patterns. In *IISWC*.

[2] Azza Abouzeid, Kamil Bajda-Pawlikowski, Daniel J. Abadi, Alexander Rasin, and Avi Silberschatz. 2009. HadoopDB: An Architectural Hybrid of MapReduce and DBMS Technologies for Analytical Workloads. *PVLDB* 2, 1 (2009), 922–933.

[3] Sibel Adali, K. Selçuk Candan, Yannis Papakonstantinou, and V. S. Subrahmanian. 1996. Query Caching and Optimization in Distributed Mediator Systems. In *SIGMOD*.

[4] Mohammed Al-Kateb, Paul Sinclair, Alain Crolotte, Lu Ma, Grace Au, and Sanjay Nair. 2017. Optimizing UNION ALL Join Queries in Teradata. (2017).

[3] Spark appears to be already moving in this direction [49].

[5] Alexander Alexandrov, Rico Bergmann, Stephan Ewen, Johann-Christoph Freytag, Fabian Hueske, Arvid Heise, Odej Kao, Marcus Leich, Ulf Leser, Volker Markl, Felix Naumann, Mathias Peters, Astrid Rheinländer, Matthias J. Sax, Sebastian Schelter, Mareike Höger, Kostas Tzoumas, and Daniel Warneke. 2014. The Stratosphere platform for big data analytics. *VLDB Journal* 23, 6 (2014), 939–964.
[6] Apache Software Foundation. 2017. *Apache Arrow.* http://arrow.apache.org
[7] Apache Software Foundation. 2017. *Apache Avro.* https://avro.apache.org
[8] Apache Software Foundation. 2017. *Apache Calcite.* https://calcite.apache.org
[9] Apache Software Foundation. 2017. *Apache Cassandra.* http://cassandra.apache.org
[10] Apache Software Foundation. 2017. *Apache Flink.* https://flink.apache.org
[11] Apache Software Foundation. 2017. *Apache Flume.* http://flume.apache.org
[12] Apache Software Foundation. 2017. *Apache Parquet.* https://parquet.apache.org
[13] Apache Software Foundation. 2017. *Apache Sqoop.* http://sqoop.apache.org
[14] Apache Software Foundation. 2017. *Hadoop.* http://hadoop.apache.org
[15] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. 2015. Spark SQL: Relational Data Processing in Spark. In *SIGMOD*.
[16] Francesca Bugiotti, Damian Bursztyn, Alin Deutsch, Ioana Ileana, and Ioana Manolescu. 2015. Invisible Glue: Scalable Self-Tuning Multi-Stores. In *CIDR*.
[17] Michael J. Carey, Laura M. Haas, Peter M. Schwarz, Manish Arya, William F. Cody, Ronald Fagin, Myron Flickner, Allen Luniewski, Wayne Niblack, Dragutin Petkovic, Joachim Thomas II, John H. Williams, and Edward L. Wimmers. 1995. Towards Heterogeneous Multimedia Information Systems: The Garlic Approach. In *RIDE-DOM*.
[18] Michael J. Carey and Hongjun Lu. 1986. Load Balancing in a Locally Distributed Database System. In *SIGMOD*.
[19] Sudarshan S. Chawathe, Hector Garcia-Molina, Joachim Hammer, Kelly Ireland, Yannis Papakonstantinou, Jeffrey D. Ullman, and Jennifer Widom. 1994. The TSIMMIS Project: Integration of Heterogeneous Information Sources. In *IPSJ*. 7–18.
[20] DataStax. 2017. *Apache Cassandra. Partitioners.* http://docs.datastax.com/en/cassandra/3.0/cassandra/architecture/archPartitionerAbout.html
[21] Amol Deshpande and Joseph M. Hellerstein. 2002. Decoupled Query Optimization for Federated Database Systems. In *ICDE*.
[22] David J. DeWitt, Alan Halverson, Rimma V. Nehme, Srinath Shankar, Josep Aguilar-Saborit, Artin Avanes, Miro Flasza, and Jim Gramling. 2013. Split query processing in polybase. In *SIGMOD*.
[23] Weimin Du, Ravi Krishnamurthy, and Ming-Chien Shan. 1992. Query Optimization in a Heterogeneous DBMS. In *VLDB*.
[24] Jennie Duggan, Aaron J. Elmore, Michael Stonebraker, Magdalena Balazinska, Bill Howe, Jeremy Kepner, Sam Madden, Dave Maier, Timothy Mattson, and Stan Zdonik. 2015. The BigDAWG Polystore System. *ACM Sigmod Record* 44, 3 (2015).
[25] Avrilia Floratou, Umar Farooq Minhas, and Fatma Özcan. 2014. SQL-on-Hadoop: Full Circle Back to Shared-Nothing Database Architectures. *PVLDB* 7, 12 (2014), 1295–1306.
[26] Avrilia Floratou, Nikhil Teletia, David J. DeWitt, Jignesh M. Patel, and Donghui Zhang. 2012. Can the Elephants Handle the NoSQL Onslaught? *PVLDB* 5, 12 (2012), 1712–1723.
[27] Minos N. Garofalakis and Yannis E. Ioannidis. 1997. Parallel Query Scheduling and Optimization with Time- and Space-Shared Resources. In *VLDB*.
[28] Ahmad Ghazal, Tilmann Rabl, Minqing Hu, Francois Raab, Meikel Poess, Alain Crolotte, and Hans-Arno Jacobsen. 2013. BigBench: towards an industry standard benchmark for big data analytics. In *SIGMOD*.
[29] Scott C. Gray, Fatma Özcan, Herbert Pereyra, Bert van der Linden, and Adriana Zubiri. 2015. *SQL-on-Hadoop without compromise.* Technical Report. IBM.
[30] Laura M. Haas, Donald Kossmann, Edward L. Wimmers, and Jun Yang. 1997. Optimizing Queries Across Diverse Data Sources. In *VLDB*.
[31] Brandon Haynes, Alvin Cheung, and Magdalena Balazinska. 2016. PipeGen: Data Pipe Generator for Hybrid Analytics. In *SoCC*.
[32] Wei Hong and Michael Stonebraker. 1993. Optimization of Parallel Query Execution Plans in XPRS. *Distributed and Parallel Databases* 1, 1 (1993), 9–32.
[33] Svilen Kanev, Juan Pablo Darago, Kim M. Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David M. Brooks. 2016. Profiling a warehouse-scale computer. *IEEE Micro* 36, 3, 54–59.
[34] Manos Karpathiotakis, Ioannis Alagiannis, Thomas Heinis, Miguel Branco, and Anastasia Ailamaki. 2015. Just-In-Time Data Virtualization: Lightweight Data Management with ViDa. In *CIDR*.
[35] Ralph Kimball and Margy Ross. 2002. *The data warehouse toolkit: the complete guide to dimensional modeling, 2nd Edition.* Wiley.
[36] Thomas Kirk, Alon Y. Levy, Yehoshua Sagiv, and Divesh Srivastava. 1995. The Information Manifold. In *AAAI Spring Symposium on Information Gathering*. 233–246.
[37] Marcel Kornacker, Alexander Behm, Victor Bittorf, Taras Bobrovytsky, Casey Ching, Alan Choi, Justin Erickson, Martin Grund, Daniel Hecht, Matthew Jacobs, Ishaan Joshi, Lenni Kuff, Dileep Kumar, Alex Leblang, Nong Li, Ippokratis Pandis, Henry Robinson, David Rorke, Silvius Rus, John Russell, Dimitris Tsirogiannis, Skye Wanderman-Milne, and Michael Yoder. 2015. Impala: A Modern, Open-Source SQL Engine for Hadoop. In *CIDR*.
[38] Donald Kossmann. 2000. The State of the Art in Distributed Query Processing. *ACM Computing Surveys* 32, 4 (2000), 422–469.
[39] Jeff LeFevre, Jagan Sankaranarayanan, Hakan Hacigümüs, Jun'ichi Tatemura, Neoklis Polyzotis, and Michael J. Carey. 2014. MISO: souping up big data query processing with a multistore system. In *SIGMOD*.
[40] Maurizio Lenzerini. 2002. Data Integration: A Theoretical Perspective. In *PODS*.
[41] Haoyuan Li, Ali Ghodsi, Matei Zaharia, Scott Shenker, and Ion Stoica. 2014. Tachyon: Reliable, Memory Speed Storage for Cluster Computing Frameworks. In *SoCC*.
[42] Ling Liu and M. Tamer Özsu (Eds.). 2009. *Encyclopedia of Database Systems.* Springer US.
[43] Lothar F. Mackert and Guy M. Lohman. 1986. R* Optimizer Validation and Performance Evaluation for Distributed Queries. In *VLDB*.
[44] Mary Tork Roth and Peter M. Schwarz. 1997. Don't Scrap It, Wrap It! A Wrapper Architecture for Legacy Data Sources. In *VLDB*.
[45] Tim Mattson, Vijay Gadepally, Zuohao She, Adam Dziedzic, and Jeff Parkhurst. 2017. Demonstrating the BigDAWG Polystore System for Ocean Metagenomics Analysis. In *CIDR*.
[46] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. 2008. Pig Latin: A Not-So-Foreign Language for Data Processing. In *SIGMOD*.
[47] Kay Ousterhout, Ryan Rasti, Sylvia Ratnasamy, Scott Shenker, and Byung-Gon Chun. 2015. Making Sense of Performance in Data Analytics Frameworks. In *NSDI*.
[48] Lin Qiao, Yinan Li, Sahil Takiar, Ziyang Liu, Narasimha Veeramreddy, Min Tu, Ying Dai, Issac Buenrostro, Kapil Surlaker, Shirshanka Das, and Chavdar Botev. 2015. Gobblin: Unifying Data Ingestion for Hadoop. *PVLDB* 8, 12 (2015), 1764–1775.
[49] Robert Kruszewski. 2016. *Catalyst: Allow adding custom optimizers.* https://issues.apache.org/jira/browse/SPARK-9843
[50] Mary Tork Roth, Fatma Ozcan, and Laura M. Haas. 1999. Cost Models DO Matter: Providing Cost Information for Diverse Data Sources in a Federated System. In *VLDB*.
[51] Richard T. Snodgrass and Ilsoo Ahn. 1985. A Taxonomy of Time in Databases. In *SIGMOD*.
[52] Mohamed A. Soliman, Lyublena Antova, Venkatesh Raghavan, Amr El-Helw, Zhongxian Gu, Entong Shen, George C. Caragea, Carlos Garcia-Alvarado, Foyzur Rahman, Michalis Petropoulos, Florian Waas, Sivaramakrishnan Narayanan, Konstantinos Krikellas, and Rhonda Baldwin. 2014. Orca: A Modular Query Optimizer Architecture for Big Data. In *SIGMOD*.
[53] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. 2009. Hive - A Warehousing Solution Over a Map-Reduce Framework. *PVLDB* 2, 2 (2009), 1626–1629.
[54] Anthony Tomasic, Louiqa Raschid, and Patrick Valduriez. 1998. Scaling Access to Heterogeneous Data Sources with DISCO. *TKDE* 10, 5 (1998), 808–823.
[55] Transaction Processing Performance Council. 2017. *TPCx-BB.* http://www.tpc.org/tpcx-bb
[56] Panos Vassiliadis. 2011. A Survey of Extract-Transform-Load Technology. In *Integrations of Data Warehousing, Data Mining and Database Technologies - Innovative Approaches*. 171–199.
[57] Panos Vassiliadis and Alkis Simitsis. 2009. Near Real Time ETL. In *New Trends in Data Warehousing and Data Analysis*. 1–31.
[58] Jingjing Wang, Tobin Baker, Magdalena Balazinska, Daniel Halperin, Brandon Haynes, Bill Howe, Dylan Hutchison, Shrainik Jain, Ryan Maas, Parmita Mehta, Dominik Moritz, Brandon Myers, Jennifer Ortiz, Dan Suciu, Andrew Whitaker, and Shengliang Xu. 2017. The Myria Big Data Management and Analytics System and Cloud Services. In *CIDR*.
[59] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *NSDI*.