A Robust and Extensible Tool for Data Integration Using Data Type Models

Andres Quiroz, Eric Huang, and Luca Ceriani
Palo Alto Research Center
Fn.Ln@parc.com

Abstract

Integrating heterogeneous data sets has been a significant barrier to many analytics tasks, due to the variety in structure and level of cleanliness of raw data sets requiring one-off ETL code. We propose HiperFuse, which significantly automates the data integration process by providing a declarative interface, robust type inference, extensible domain-specific data models, and a data integration planner which optimizes for plan completion time. The proposed tool is designed for schema-less data querying, code reuse within specific domains, and robustness in the face of messy unstructured data. To demonstrate the tool and its reference implementation, we show the requirements and execution steps for a use case in which IP addresses from a web clickstream log are joined with census data to obtain average income for particular site visitors (IPs), and offer preliminary performance results and qualitative comparisons to existing data integration and ETL tools.

1 Introduction

The era of Big Data is upon us, and with it, the opportunity to use new and vast potential sources of information that until recently were not easily exploitable because of limitations in storage and processing power, and because of system isolation and even lack of foresight (data being thrown away because it was not deemed useful). Today, we are recognizing the value of collecting, storing, sharing, and integrating data from heterogeneous sources such as web logs, social media activity, and all sorts of open demographic data files and databases in local systems and on the web.

Given that the tendency is to keep data because of the expectation that it will be useful, and not necessarily because of an immediate need, the process of extracting and transforming data from sources tends to be delayed and to leave more data stored closer to its original (raw) form. Extracting, transforming, and integrating multiple of these heterogeneous raw sources are therefore increasingly being done in a more on-demand, and ongoing basis, for data mining, exploring new hypotheses, or supporting new applications.

Palo Alto Research Center engages companies in various industries with research-oriented, exploratory projects aimed to solve some of their strategic big data analytics problems. Many of our clients with problems from medical fraud detection to retail found the extract, transform, and load (ETL) process a significant bottleneck due to the variety of messy data sets. For a Fortune 500 company we explored analytics applications on their online purchases with hundreds of millions of rows, and a clickstream data set of over 30TB. Though some vertical applications were built, much of the ETL code was one-off and consumed the majority of our time. In order for PARC to repeatedly provide data discovery and analytics services, the Xerox Innovation Group funded this project to automate the data integration process so that analytics services requiring data integration may be scalable, and repeated with high agility. Our work in progress, HiperFuse, is a tool designed to address the challenges due to the variety in structure and level of cleanliness of raw data sets by providing automation in the data integration process and leveraging four key capabilities:

1. Providing a declarative interface for the simple specification of the initial and goal states of data,
2. Modularly separating models of the data types and transformations from the operational code,
3. Leveraging a library of domain-specific models of data types and transformations, which are easily extended, and
4. Using a planner and scheduler to find and optimize a parallel data integration plan, minimizing latency.

HiperFuse has been designed for 1) the common user, usually an analyst or data scientist, who only needs to express the desired output data set, counting on pre-existing data type models and transformations; and 2) the engineer who will extend HiperFuse's library of domain-specific models. Given a domain-specific library, a user may declare the current and desired goal states of the data, after which such directives are then interpreted by a data integration planning engine. The planner generates a plan by using known or inferred data types and data transformation actions. The resulting plan is mapped to executable scripts which may be on the local file system, a SQL database query, or even a MapReduce job on HDFS. These scripts are then executed, producing the integrated data set. As shown in Figure 1, HiperFuse splits its execution between the local machine, where a user inputs a query via the declarative interface and where the planner generates and orchestrates an

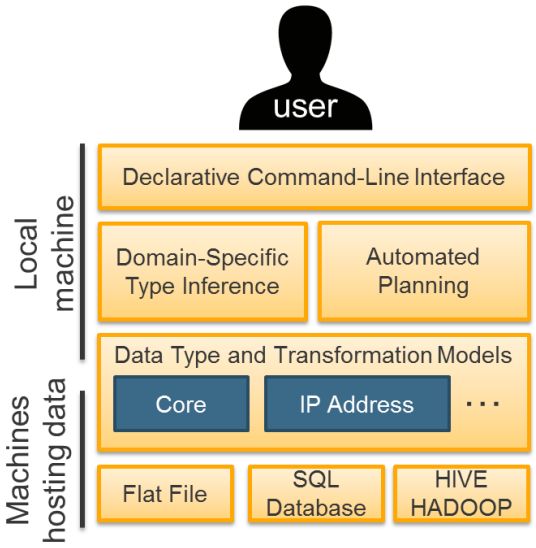

Figure 1: Top level architecture of HiperFuse

execution plan to carry it out, and the systems housing the data, where the final executed scripts, including some data statistics and type inference actions, are run.

Typical code written for ETL often mixes different logical layers of Figure 1, which are separate in HiperFuse's architecture. A side effect of this is that the actual models of the data types and transformations can be extended and reused without having to duplicate and adapt previously written ETL code. This type-centric approach to data integration also has other advantages. One advantage of modeling data operations and transformations around higher-level, domain-specific data types, besides promoting code reuse, is being able to automatically detect and notify when and what operations are possible on any given data set being analyzed. For example, in addition to tagging a column that contains IPs in a file with the IP data type, the result of type inference can include descriptions of existing operations that have been previously written for the IP type, such as masking and group decomposition. Additionally, generic operations such as equality comparisons and set membership can be automatically adapted at runtime while providing type-specific functionality depending on the data type detected. The end user thus only needs to know about the generic operations, and let HiperFuse's type inference system make the necessary adaptations to the input, if possible, to make it executable (and if not possible, request missing type or operation definitions). These advantages facilitate the task of a user and minimize the amount of one-off or boilerplate code that needs to be manually written.

Related Work

In general, data integration seeks to make multiple data sources queryable from a single unified schema. Global-As-View vs. Local-As-View are two strategies for describing data source and schema integration, where one maps the local schema to the global schema or one maps the global schema to the local one, respectively (Ullman 2000; Halevy 2001). HiperFuse takes a hybrid approach and thus, faces related challenges, such as robust data type inference, mechanics and efficiency of (and ability to express) raw data joins, and entity resolution and disambiguation (although the latter challenge is generally out of our scope).

The main limitations of popular tools are as follows. First, many integration tools require imperative programming – i.e. the user must decide what data transformations to perform as well as their ordering. Second, the lack of sophisticated models for data types keeps the process manual and time consuming. Lastly, there are very poor ETL-workflow optimization algorithms available for this problem. We review some popular tools in more detail here.

As the volume of data collected increases, many ETL tasks involve data in HDFS. Tools built on top of Hadoop provide a visual and workflow-oriented interface for ETL, as well as metadata management for data integration. Examples of such tools and vendors popular today are Talend, Informatica, Red Point, and Ataccama. They provide multiple connectors for moving data between multiple platforms, and allow users to inspect and describe their data, define transformation workflows, and execute MapReduce ETL jobs. Many of these tools also provide a graphical user interface to aid in programming. The data flow is explicitly represented, and the user drags and drops connectors to data sources and operations. Tools such as Stanford Data Wrangler, Trifacta and Datameer supplement such graphical interfaces with some basic statistics over the data as well, but though the claim of many of these tools is that no programming is required, in fact the user is still expressing an imperative program whose code cannot be easily reused when the data set quality, schemas, and sources change.

Systems such as Cascalog reduce the amount of code required for ETL using functional programming and Datalog principles. However, one still specifies inference rules to recognize data types tied to particular data sets, and its rule-rewriting mechanism doesn't minimize latency of the ETL plan across a variety of data sources. IBM TSIMMIS (Chawathe et al.) separates data models from the operational ETL code, but also depends on rule rewriting which also does not generate ETL plans that minimize completion time explicitly. MiniCon (Pottinger and Halevy 2001) uses combinatorial search instead of rule rewriting, but does not model heterogeneous data store performance and explicitly minimize completion time.

There are systems that mitigate the need for semantic knowledge of the data type. The Hadoop core tools (Hive and Pig) defer data type validation until runtime, but data type labeling must be done explicitly and manually for new queries. Any domain-specific operations require writing user-defined functions that have hard-coded constraints and semantic knowledge for the specific problem, making them hard to reuse. Take the IP address use case described below in Section 2. In Hive, there is simply no way currently to express the requirement as a single high-level query, with operations at the level of the IP address field. Instead, the IP address should be either transformed into integer form or decomposed into its constituent groups, and then linked

via multiple queries to the remaining files. Though the task is somewhat more straightforward in Pig, resulting in a single compact script, the combination of steps from the script would be very difficult to encapsulate and reuse.

MySQL and similar systems compute basic statistics that infer the column types but the inference is limited to low-level data types and is highly sensitive to minor errors in the data. Even a small percentage of malformed rows in a large data set could cause millions of exceptions, crippling performance and resulting in the labeling of all fields as strings. Finally, the data types inferred by these tools lack the domain-specific meaning to enable higher-order functionality.

Finally, note that little work exists for benchmarking data integration, due to the difficulty of quantifying data set heterogeneity. Various efforts have attempted to create a set of metrics for benchmarking, including the Workshop on Big Data Benchmarking, which began only recently in 2012 (Baru et al. 2013a) due to the combined efforts of the Center for Large-Scale Data Systems at UCSD, the Transactions Processing Performance Council, and the Big Data Top100 benchmarking community (Baru et al. 2013b). The Transactions Processing Performance Council has set forth the specifications for benchmarks for the Transform and Loading phases of ETL, but do not attempt to address the extraction phase (Poess et al. 2002). Thus, the problem of automating data integration is not fully addressed by this benchmark, as the TPC-DS benchmark makes assumptions about the initial and destination forms (i.e. tables conforming to a normalized star schema) that are too narrow. A more comprehensive survey of various industrial tools, including many of these observations, has been published by Gartner (R. L. Sallam).

2 HiperFuse: High Performance Data Integration

In the following subsections we will describe each of the various components in HiperFuse's architecture (Figure 1) in detail and describe how they interact. We first introduce a pedagogical use case in which IP addresses from a web clickstream log are joined with census data to obtain average income for particular site visitors; this use case will help illustrate various of the key functionalities of HiperFuse.

Example: Linking IP Address to Median Income

Consider, for example, the need to integrate three data sets based on mapping IP addresses to IP address blocks, IP address blocks to zip codes, and zip codes to median household income. A web clickstream log with originating IP address on each row resides in HDFS, while a data set that maps IP address blocks to zip codes resides on the local file system. A data set from the U.S. census that maps zip codes to median household income also resides in the local file system. Samples of these data sets are shown in Figure 2. Using standard tools, an experienced analyst might:

- Cleanse, restructure and validate the two data sets on the local file system using UNIX utilities,
- Create SQL schemas in an RDBMS and import the data,

Apache clickstream log

```
199.120.110.21 - - [01/Jul/1995:00:00:11 -0400] "GET..."
burger.letters.com - - [01/Jul/1995:00:00:12 -0400] "GET..."
burger.letters.com - - [01/Jul/1995:00:00:12 -0400] "GET..."
205.212.115.106 - - [01/Jul/1995:00:00:12 -0400] "GET..."
d104.aa.net - - [01/Jul/1995:00:00:13 -0400] "GET..."
129.94.144.152 - - [01/Jul/1995:00:00:13 -0400] "GET..."
```

IP address to zip code data (Maxmind)

67277040	67277047	4.2.144.240	4.2.144.247	76092
67277048	67277055	4.2.144.248	4.2.144.255	45202
67277056	67277215	4.2.145.0	4.2.145.159	""
67277216	67277247	4.2.145.160	4.2.145.191	75038
67277248	67277407	4.2.145.192	4.2.146.95	""
67277408	67277439	4.2.146.96	4.2.146.127	75038

Zip code to income data (US Census 2000)

00637	0453044	5931	3372	3299	1940	1864	76
00638	0453045	3408	2128	2044	1317	1233	84
00641	0453046	6183	3658	3306	1657	1545	112
00646	0453047	8866	5151	5045	3539	3278	261
00647	0453048	852	536	520	319	319	0
00650	0453049	2295	1363	1309	862	804	58

Figure 2: Illustrative example

```
1 column("hdfs://web_access_log", 1, "client_host")
2 range("file:///ipblock2zip.tsv", [3, 4], "ip-block")
3 column("file:///ipblock2zip.tsv", 5, "zip1")
4 column("file:///census.tsv", 1, "zip2")
5 column("file:///census.tsv", 40, "income")
6 member_of("client_host", "ip_block")
7 equals("zip1", "zip2")
8 # Output
9 make_tsv("file:///ip2income.tsv", "client_host", "income")
```

Figure 3: Example HiperFuse query for obtaining the average income for a set of IP addresses

- Write SQL queries to join and create a new data set mapping just IP address blocks to median income,
- Move the result from the SQL database into HDFS,
- Create HIVE metadata schemas in which all fields are interpreted as strings or integers for all data sets,
- Cleanse and validate the data in HDFS, creating user defined functions and intermediate tables with specialized fields to handle the necessary joins,
- Perform the final join, and output the mapping of IP address to median income.

There are various tools previously mentioned that one can substitute for some steps, but the main disadvantages of this process are that the code that is written is not very reusable because the programs tie together many aspects specific to this ETL instance: structure and cleanliness of all three data sets, performance characteristics of the storage platforms, schema and fields of the data sets, desired destination, and semantic knowledge of a domain-specific record linkage.

Declarative Interface

Leveraging a planning engine, HiperFuse only requires a set of declarative directives provided in a file telling it the starting state of the data in terms of structure, and the goal state of the data. Figure 3 shows the code required to integrate the three data sets in our example.

```
1  column("hdfs://web_access_log", 1, "client_host")
2  range("file:///ipblock2zip.tsv", [3, 4], "ip-block")
3  column("file:///ipblock2zip.tsv", 5, "zip1")
4  column("file:///census.tsv", 1, "zip2")
5  column("file:///census.tsv", 40, "income")
6  ip_type("client_host")
7  zip_type("zip")
8  int_type("income")
9  ip_type(3)
10 ip_type(4)
11 zip_type(zip1)
12 # Conditional operations
13 member_of("client_host","ip_block")
14 equals("zip1", "zip2")
15 # Output
16 make_tsv("file:///ip2income.tsv", "client_host", "income")
```

Figure 4: Executable HiperFuse query after type inference

```
{columnName:client_host, minLength:2, maxLength:15, avgLength:15,
alphabet:" 0123456789", extraneousChars:"#\u0026\u0027*,/:@HKNOSTUWY_jqz"}
```

Figure 5: Cleansing output for the *client_host* field

Here, lines 1-5 name five data types called *client_host*, *ip-block*, *zip1*, *zip2*, and *income*. These predicates also point to the corresponding files and columns where the data resides. Note that *ip-block* logically groups two separate columns from one file into a single entity reference. Lines 6 and 7 state the relations among these data fields (e.g. that values in column *client_host* fall in the ranges of values in *ip-block*), which will be used by HiperFuse as join conditions during the execution of the query. Finally, line 9 says the output is a local TSV file with columns *client_host* and *income*.

Note that no type information needs to be given in the input query – simply the required fields in the data and their relation via the generic operations `member_of`, which indicates set or range membership, and `equals`. As will be subsequently described, HiperFuse's type inference can use a set of models that contain definitions for IP address and zip code to produce the executable query from Figure 4, in which the inferred type declarations added by the inference component have been highlighted. With the given type information, HiperFuse can appropriately parse the fields in the data and apply the type specific definitions of the `member_of` and `equals` operations.

Data Cleansing

Data cleansing and structuring happens as the first step when ingesting raw data. The output of this step are tab-delimited data files in which there are no control characters (besides tabs to separate fields), no rows with mismatched quote characters, no field values with tab characters within them, and where every row has the same number of fields. During this process, various aggregate statistics also are computed and are made available for inference steps elsewhere in HiperFuse. Figure 5 shows an example of one record in the output of the cleansing stage, containing these statistics for the *client_host* field in the clickstream file.

Because even the delimiters and the number of fields may not be initially clear, HiperFuse computes statistics such as the histogram of the number of fields that result from segmenting the rows using various delimiters, the alphabet used in each column (excluding characters that only occur in a small percentage of the column values), and other simple statistics over the string lengths in each column. In the data sets we used, website attacks had injected binary characters and malformed rows into the files. Such errors are cleansed and removed by HiperFuse, but indicators for them can still be found in the statistics output: unexpected field lengths and extraneous characters.

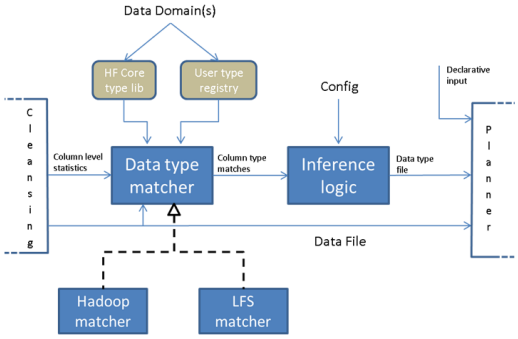

Figure 6: Type inference framework

Type Inference

HiperFuse provides a robust and extensible framework for type inference that leverages both core domain knowledge and application-specific knowledge. The components and workflow for this framework are illustrated in Figure 6.

As a general approach, the models from the *type library* are tested against every value in a column, and a *data type matcher* generates a histogram showing how many rows match a given data type. The resulting histograms are analyzed for peaks that in many cases easily stand out, although the actual *inference logic* used to analyze the histogram information and provide the actual type inference output is implemented as a separate module with configurable parameters. In our IP address example, one may decide that if 95% of the row values for a given column match the IP address model, then HiperFuse will conclude that column to be of type IP address, and automatically add the `ip_type` statement to the original declarative program. This combination of histogram analysis along with pluggable inference logic makes data type inference in HiperFuse very robust to the presence of errors in raw data and flexible in its interpretation of fields with mixed or ambiguous data.

HiperFuse maintains data type model information both in an integrated core type library and as a registry of user-defined types. The core library is meant to be optimized for performance, as it may exploit relationships among known types to more efficiently test a set of data type hypotheses. The type registry, on the other hand, is designed for easy extensibility and thus contains a set of black box type detectors, which could implement simple models such as regular expressions, but also complex and flexible models such as those produced by machine learning methods. HiperFuse's core data types models will cover strings, integers,

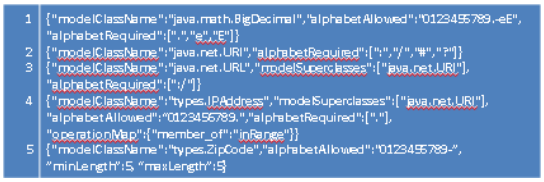

Figure 7: HiperFuse type registration file

Method	Time (sec)
Full scan	1464
Perfect fore-knowledge	206
Bandits	107
Bandits and column statistics	61

Table 1: Runtimes

and dates/times, as do most current tools that provide basic type inference, while user-defined models can be registered for recognizing IP addresses, IP address blocks, and zip codes as required by the use case.

Because the architecture is designed for scalability across additional domains (for example, geographic data domain, financial/banking domain, etc.), computational complexity is a concern as the data type library increases in size. The development of HiperFuse will therefore continually incorporate domain-specific models directly into its core type library, as long as they are general enough to use across multiple data sets in different applications. IP addresses and URLs are good examples of types that HiperFuse will include directly as core types for a web data domain.

Another technique to reduce complexity despite a large type library in HiperFuse is to prune the space of candidate data types that must be tested. Using the output from the data cleansing stage, HiperFuse can remove candidate data types that are not compatible with this information. A prerequisite for this is a type model registration file, such as the example shown in Figure 7. Pruning is accomplished by using the alphabet allowed, alphabet required, and/or length constraints that can be specified for each data type, given the output of the cleansing stage (Figure 5). For example, since the alphabet for *client_host* contains the dot (".") character, the zip code model, which does not include the dot in its alphabet allowed constraint, is never tested for matches in that field.

Even with pruning, the volume of data records and the number of fields in big data files pose a computational challenge. That is, supposing one could perfectly guess and prune down to a single correct model for each field in the data to test, testing the model against all values in the data can still be computationally expensive. This is especially true for user defined types that cannot be expected to benefit from the hierarchical pruning that can be done using the parse tree from the core type library.

In the data type matcher, as models are tested against each field value, a probability distribution for the aggregate probability of success of each model is updated. Before each new test, these probability distributions are sampled for each model, so that the new test is done only if the sampled probability value shows uncertainty in the success of the test (i.e. if p is the probability of success, a trial will be run for $l \leq p < u$, where l and u are a lower and upper bound respectively). The intuition of the approach is that, when there is a good fit of a model to a column, most tests of the model will succeed, and otherwise most will fail. New values therefore only need to be tested when there is uncertainty because of an unclear majority of successes or failures. However, the sampling of the probability distribution always allows for the possibility of gathering new information in cases of skewed distributions of values in particular fields, especially for large data sets. This optimization, which we call *selective model invocation*, is inspired by the Bayesian bandits problem, and differs from current approaches that sample a subset of values to be tested a-priori.

Table 1 shows the performance impact of the complexity reduction methods described above. It shows the timing of HiperFuse's type inference run on Hadoop on a single block of data from the `ip2zip` file, containing 12 data fields. Type inference was run using only user defined types in the type model registry, for 10 data types. The last row shows the effect on performance of running selective model invocation (Bandits) and pruning the model space using the column statistics from the data cleansing stage. Notice that it is a 70% improvement on the result that could be obtained with perfect fore-knowledge (testing a column only against the correct type model). Of course, as can be seen from the time for a full scan, running all type models on all columns is clearly not feasible.

HiperFuse can be extended with different implementations of data type matchers, i.e the modules that compute the type match statistics, depending on where data is located. A local file system (LFS) implementation may be able to exploit the fact that it can see all of the data for a single data set locally, while such an assumption generally will not hold on HDFS, requiring different approaches for aggregating statistics and employing different performance optimizations.

Data Integration Planner

After the user inspects and possibly alters the augmented declarative program, a custom data integration planner is invoked. Our planner explicitly represents each field of a data table within files, as well as the properties associated with those fields. Properties may include "sortedness", type information, or other constraints. In contrast to modeling actions with fixed parameters over a set of predicates, we have found it more natural to model actions in data integration as operations with set-based semantics such as those explicitly represented by relational algebra.

We have had very little success at encoding our domain in the Planning Domain Definition Language, a general language understood by many state-of-the-art planning and scheduling software (Fox and Long 2003). The initial and goal states, along with the data transformation models constitute the facts and actions that encapsulate a canonical planning problem. The domain closely models the underlying data transformation operations and the data types. The main challenges in modeling UNIX operations such as cat,

cut, and sort in PDDL are known modeling challenges in this domain, such as the duplication or creation of new files, allocating new file descriptors to facilitate piping, and representing actions with variable number of parameters.

In this model, we use predicates to represent the availability of file descriptors. We enumerate a finite set of constants to allow the planner to pick a new file descriptor as needed. Because the cut command represents selecting a subset of columns, we need a way to encode a variable number of columns chosen. Thus a helper action with zero cost becomes enabled for the duration of the cut command, which adds a specific column to this set. Likewise, when a column is selected, an action is enabled that copies to the new column metadata properties from the selected column. Finally, we distinguish between streaming vs. blocking operations by stating whether the output file descriptor is available at the start of the action or at the end.

We tried both CPT-YAHSP (Dreo et al. 2014) and Temporal Fast Downward (Eyerich, Mattmüller, and Röger 2009), two state-of-the-art temporal planners that have participated in the International Planning Competitions in different years, but neither was able to solve a simple join problem. In some instances, logical bugs pruned the space of all solutions, and in others, grounding all of the actions caused the planner to hang. This is easy to see in that we are modeling join actions which have as parameters multiple input files, multiple join columns from each file to join upon, multiple columns to choose from as the output of the join, and also must copy over any subset of properties that hold true for each of these data columns. Many of these parameters are drawn from a potentially infinite namespace. We are currently examining backward-chaining lifted planners such as VH-POP (Simmons and Younes 2011), but lifted planners have not received as much recent attention as ones which assume grounding, in addition to the fact that many of the temporal features of the planning language which we rely on are unsupported.

3 Conclusions and Ongoing Work

Currently, the separate modules of HiperFuse have been implemented and tested with the expected inputs and outputs for each piece, but not yet integrated. The inference and data cleansing components have been implemented on both the local file system as well as in Hadoop, and demonstrated to correctly identify the fields of the multiple files of the IP address use case described herein, yielding the preliminary performance results shown. For contrast, code was written for running the same use case using both Hive and Pig, which proved to be more cumbersome because of the need to manually define schemas using low level types, the lack of expressiveness of the query language that led to complex one-off scripts or multiple queries, and the extra boilerplate code needed for user defined functions.

As the project continues, we will integrate an end-to-end system for the IP-to-income use case, and then expand to additional use cases in different domains to show generality (for example, performing a join on timestamps or geolocation within a certain proximity given transportation and/or social media data). The intent is to continue to extend the system with core model libraries for different domains and to add the capability to perform semantic type analysis to obtain aggregate column properties, such as key columns, categorical vs quantity data, etc. We will also evaluate the accuracy of these models for different applications and the total cycle time of data integration in the different use cases when compared to other tools that do not offer type inference or that employ it in a more limited/different way. It will also be imperative to design and optimize on the different data platforms generic join operations for the core types, given the often intractable cost of cross joins on big data.

References

Baru, C.; Bhandarkar, M.; Nambiar, R.; Poess, M.; and Rabl, T. 2013a. Setting the direction for big data benchmark standards. In Nambiar, R., and Poess, M., eds., *Selected Topics in Performance Evaluation and Benchmarking*, volume 7755 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg. 197–208.

Baru, C.; Bhandarkar, M.; Nambiar, R.; Poess, M.; and Rabl, T. 2013b. Setting the direction for big data benchmark standards. In *Selected Topics in Performance Evaluation and Benchmarking*. Springer. 197–208.

Chawathe, S.; Garcia-Molina, H.; Hammer, J.; Ireland, K.; Papakonstantinou, Y.; Ullman, J.; and Widom, J. The tsimmis project: Integration of heterogeneous information sources.

Dreo, J.; Saveant, P.; Schoenauer, M.; and Vidal, V. 2014. Divide-and-evolve: the marriage of descartes and darwin. In *Booklet of the 7th International Planning Competition*.

Eyerich, P.; Mattmüller, R.; and Röger, G. 2009. Using the context-enhanced additive heuristic for temporal and numeric planning. In *Proceedings of the 19th International Conference on Automated Planning and Scheduling, ICAPS 2009, Thessaloniki, Greece, September 19-23, 2009*.

Fox, M., and Long, D. 2003. Pddl2.1: An extension to pddl for expressing temporal planning domains. *J. Artif. Int. Res.* 20(1):61–124.

Halevy, A. Y. 2001. Answering queries using views: A survey. *The VLDB Journal* 10(4):270–294.

Poess, M.; Smith, B.; Kollar, L.; and Larson, P. 2002. Tpc-ds, taking decision support benchmarking to the next level. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, SIGMOD '02, 582–587. New York, NY, USA: ACM.

Pottinger, R., and Halevy, A. 2001. Minicon: A scalable algorithm for answering queries using views. *The VLDB Journal* 10(2-3):182–198.

R. L. Sallam, e. a. Magic quadrant for business intelligence and analytics platforms (online). http://www.gartner.com/technology/reprints.do?id=1-1QLGACN&ct=140210.

Simmons, R. G., and Younes, H. L. S. 2011. VHPOP: versatile heuristic partial order planner. *CoRR* abs/1106.4868.

Ullman, J. D. 2000. Information integration using logical views. *Theor. Comput. Sci.* 239(2):189–210.