Tracking and Visualizing Provenance using PostgreSQL

Alexander Rush (srush@mit.edu)
Nirmesh Malviya (nirmesh@mit.edu)

December 9, 2010

Abstract

Many scientific and business applications require keeping detailed track of the origin of data items. Such history is referred to as provenance or lineage of data. The most basic data provenance operations involve tracking which data items an existing data item was derived from (backward data provenance) and what all data items were derived from a given data item (forward data provenance).

In this work, we modify the internals of PostgreSQL, a popular open source database, to capture data provenance for a subset of SQL DDL. We have also developed an interactive web interface to query and visualize forward and backward provenance for data items in different databases.

1 Introduction

XXX: outlines added for all sections, need to fill in details everywhere

A large number of applications such as scientific data management, data integration, information extraction and business analytics involve processing base data to generate a large amount of new data derived from the base data. Capturing which data items contributed to creation of a new data item is important in the face of issues like questionable quality of base data, potential uncertainty associated with it, as well as the authority and trust assessment of the user performing the operations. We see that if data history is not traced in any form as the operations are performed, it would be impossible to track what future tables potential errors in early stage data collection may have propagated to. Tracking down errors and updating all tuples derived from an erroneous data item can be prohibitively costly or even impossible in the absence of this historical information.

The term *provenance* refers The problem of storing and quering previous data history is known as *provenance*. Recent work in this area includes the Trio/LIVE system and the Panda project. However, neither of these projects handle backward provenance efficiently or provide language support for querying provenance.

This lack of historical record can make database systems impractical for scientific research.

One practical issue with available database systems is the difficulty of tracing data history.

Our goal is to improve upon existing provenance systems, focusing on the SciDB project. The SciDB proposal [11] lists three requirements for a useful provenance system -

- For any data element, we would like to recover the derivation history.
- If a data element is updated, we would like to trace forward to see other effected elements.
- At any point, we would like to reproduce the construction of the current data.

In particular, we hope to respond to the challenge given by [4].

> Recording the log is easy. The hard part is to create a provenance query language and an efficient implementation.

Contributions:

Implemented the ability to track provenance inside postgresql

Implemented a visualizer that makes querying forward and backward provenance easier.

don't support stored procedures , they are black boxes, and unless something useful is known about the underlying functions, we anyway do not get to know too much about them.

2 Background

There are two types of provenance queries, forward and backward provenance. In forward provenance, given some data element, we ask what data elements were produced from it and what processing operations were applied to it. In backward provenance, given some data element, we ask where it originally came from and what processing led to its creation.

Fig **??** shows examples of these two queries. We assume that our original data is $s1$, it leads to $s2$ and $s3$. In turn, $s4$ is derived from $s3$. Our forward query from $s1$ leads to each of the other data elements, while the backward query from $s4$ leads back to the root.

In the LIVE provenance system built as part of the Trio project, the database stores a back pointer for each derived data element. In our example, it would store $(s1, s2)$, $(s1, s3)$, and $(s3, s4)$ explicitly. It can then perform very efficient backwards queries by walking along this tree. Unfortunately, this technique requires storing a derivation pair for each derived and its parents, which can be very expensive.

The SciDB proposal suggests a different way to implement provenance. For forward queries, the propose using the databases version control system and following forward deltas. They note that backward provenance is more difficult to implement. Two possible methods they mention are to also include backward deltas as part of version control or to implement an algorithm which can reverse the queries from the log.

The other open quesion surrounding provenance is how to include provenance queries within SQL. There have been relatively few proposed solutions for this problem. The LIVE system implements a keyword "valid at" which queries a data element at a specific revision. For instance the query:

```
SELECT * FROM <table-name> VALID AT <revision-number>
```

This query returns the rows from a table at a specific revision number. This syntax allows a query to access a revision, but does not allow us to specify a provenance query the directly.

3 Related Work

[2]

[10]

[9]

[5]

[7]

[3]

[1]

- Trio / LIVE [12, 8]

  Implements a technique for backward provenance by connecting derived tuples to the previous tuples. Unfortunately, this technique has very high space complexity and may not scale in practice.

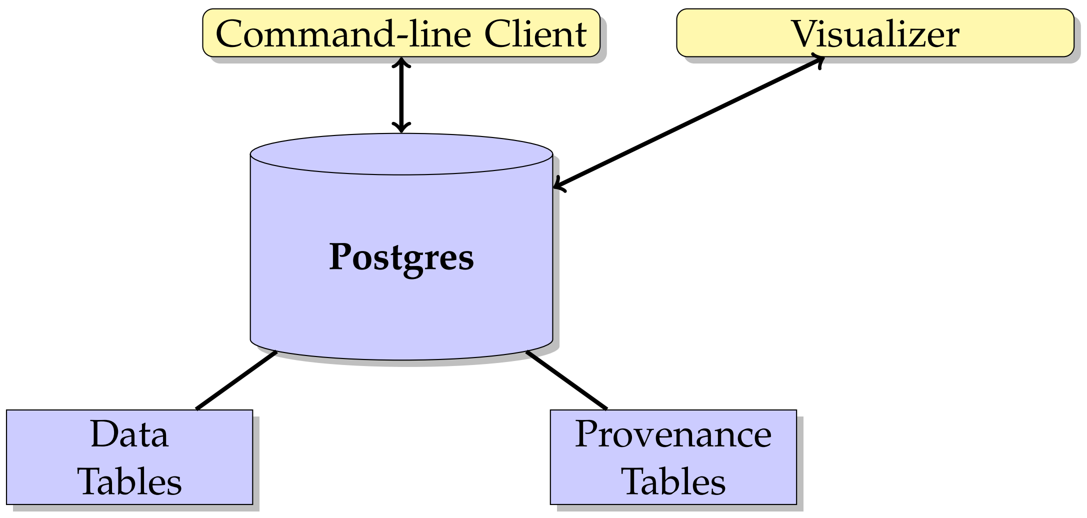
- Panda [6]

  Panda is an early stage proposal for a general provenance system for data. This system does not propose a practical algorithm or language features for provenance.

- SciDB [4]

  The SciDB system proposes a provenance system. Current proposals have low space complexity but with non-trivial query time.

4 System Architecture



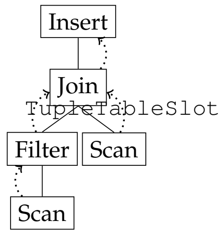


Figure 1: Provenance information is propogated with tuples through the execution plan.

5 Tracking Provenance inside PostgreSQL

The main challenge of implementing provenance is maintaining source information at a tuple granulatity, while still remaining consistent with the general structure and invariants of the Postgres executor. In this section, we describe how this data is preserved during the execution of an arbitrary SQL query.

5.1 Executor

Since we are concerned with tuple-query level provenance, we only need to preserve information from a single query. We assume that this origin information can always be read directly off of base-level tuples at the beginning of a scan [1]. When scanning a base-level tuple, we collect the table id, primary key values, and any other provenance info and pack it into a `ProvInfo` structure. The goal of this work is to route `ProvInfo` through the executor from scan to insert.

From a birds-eye view, the executor consists of a collection of node classes, one for each type of plan node. At initialization, it creates a tree of objects representing the query plan given by the optimizer. Each node has its own state and memory context and they communicate with each other through an iterator interface. The payload of this interface is the `TupleTableSlot` structure, which abstracts over various internal tuple representations. In order to pass information bottom-up to higher nodes, we append the `ProvInfo`s for the current tuple onto this slot.

```
struct TupleTableSlot {
  ...
  List <ProvInfo> provinfos;
}
```

Since all nodes communicate through this common interface, provenance information is always preserved through inter node channels. Unfortunately, a node itself may perform arbitary operations on the tuples it receives. We cannot hope to preserve provenence through all nodes, for instance stored procedures, but we can keep this information for standard SQL queries.

[1]This assumption breaks temporary tables, but these can be handled has a special case since there is enough additional information in the `pg_prov` table.

Header
TupleBody
Provenance

Figure 2: The modified `MinimalTuple` tagged with provenance infomation.

5.1.1 Scans, Filters, and Projections

The most basic nodes in a query plan are the ones that receive a single input tuple and optionally output a single tuple, e.g. scans, filters, and projections. Postgres implements this style of node as a projection directly on `TupleTableSlot`. We simply modify this projection function to preserve the `ProvInfo` from incoming tuple.

5.1.2 Joins: Hashing and Sorting

Joins follow a similar interface as scans, they take in a tuple slot and output projected tuples; however internally, they directly access the internal contents of the tuple. Both hash join nodes and sort merge join nodes first convert their input into a set of `MinimalTuples` before performing the joins.

The `MinimalTuple` structure is are direct representation of the tuple data. It is designed to use up minimal memory in cases where many tuples are in kept main memory. When a join is successfully performed, the resulting fields are projected out directly from the `MinimalTuple` representation.

In order to preserve provenance, we need to tag every minimal tuple with it provenance information. Fig 5.1.2 gives our new minimal tuple representation. In many cases this provenance information will be negligible, but it could be a source of significant slow down. We explore this furthur in Section **??**.

This extra tag is the only modification needed to handle joins. When we convert a tuple from a `TupleTableSlot` to a `MinimalTuple`, it is tagged with this information. To internal hashing, sorting, and other operations on minimal tuples this information opaque.

5.1.3 Aggregates and Grouping

The final important special case is for aggregate and grouping nodes. Postgres supports general aggregate operations and so it contains special structures for maintaining aggregate state that break our tuple assumptions. To handle aggregates conservatively, we assume that every tuple that is scanned as part of an aggregate function should be a part of the provenance information, even if it is ignored in practice. To implement this aggregation strategy, we augment the internal state maintained for each aggregate group to also include the provenance. When the aggregation is advanced by one tuple, we add this tuple's provenance information to the state information.

`pg_prov`

Attr	Type
origin_tableid	OID
dest_tableid	OID
origin_primary_key	ARRAY
dest_primary_key	ARRAY
command_id	ID

`pg_provcommand`

Attr	Type
id	ID
sql	TEXT

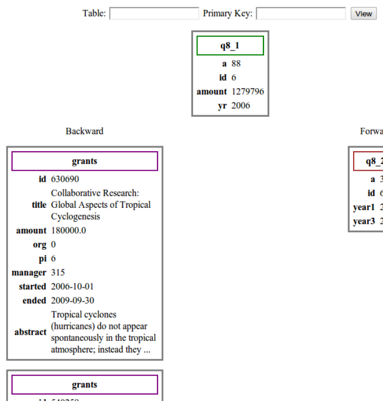


Table: Primary Key: View

q8_1

a	88
id	6
amount	1279796
yr	2006

Backward

grants

id	630690
title	Collaborative Research: Global Aspects of Tropical Cyclogenesis
amount	180000.0
org	0
pi	6
manager	315
started	2006-10-01
ended	2009-09-30
abstract	Tropical cyclones (hurricanes) do not appear spontaneously in the tropical atmosphere; instead they ...

grants

Forward

q8_2

a	3
id	6
year1	2004
year3	2006

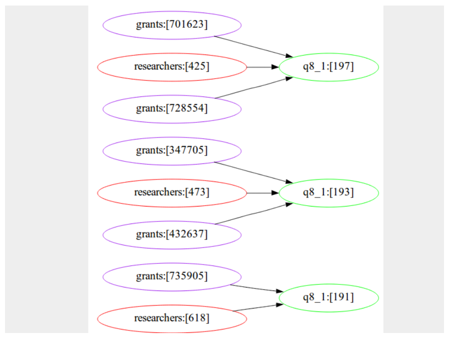


Figure 3: A web-based visualization tool running on top of pg_prov. (a) Interactive provenance viewer (b) Provenance graph display.

5.2 Insertion of Provenance

The insertion node has the task of finalizing the provenance. It first inserts the tuple (so that it has a primary key), and then write the tuples `ProvInfos` to storage. Provenance storage is implemented as a Postgres catalog table called `pg_prov` shown in Fig 5.2. The schema represents the information as a directed graph between tuples. Using the catalog for storage is convenient for analysis and visualization, and it opens up the possibility of a special syntax for querying provenance directly.

6 Visualizing Forward and Backward Provenance

To further explore the potential uses of provenance queries, we built a front-end visualization tool. The visualizer has no knowledge or connection to the internal provenance system; it reads data from the catalog tables directly.

The tool lets users view the provenance graph in two modes. In interactive mode, the user can specify a tuple by table and primary key and see it data and forward and backward provenance. The tool makes it easy move forward and backward in the graph structure. In graph mode, the user sees a compressed version of the full provenance graph in one view.

7 Conclusion and Future Work

Just reword the abstract.

References

[1] Practical lineage tracing in data warehouses. In *Proceedings of the 16th International Conference on Data Engineering* (Washington, DC, USA, 2000), IEEE Computer Society, pp. 367–.

[2] Buneman, P., Khanna, S., and chiew Tan, W. Data provenance: some basic issues. In *In Foundations of Software Technology and Theoretical Computer Science* (2000), Springer, pp. 87–93.

[3] Chapman, A. P., Jagadish, H. V., and Ramanan, P. Efficient provenance storage. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data* (New York, NY, USA, 2008), SIGMOD '08, ACM, pp. 993–1006.

[4] Cudré-Mauroux, P., Kimura, H., Lim, K., Rogers, J., Simakov, R., Soroush, E., Velikhov, P., Wang, D., Balazinska, M., Becla, J., et al. A demonstration of SciDB: a science-oriented DBMS. *Proceedings of the VLDB Endowment 2*, 2 (2009), 1534–1537.

[5] Glavic, B., and Dittrich, K. Data provenance: A categorization of existing approaches .

[6] Ikeda, R., and Widom, J. Panda: A system for provenance and data. In *Proceedings of the 2nd USENIX Workshop on the Theory and Practice of Provenance (TaPP10)* (2010).

[7] Karvounarakis, G., Ives, Z. G., and Tannen, V. Querying data provenance. In *Proceedings of the 2010 international conference on Management of data* (New York, NY, USA, 2010), SIGMOD '10, ACM, pp. 951–962.

[8] Sarma, A., Theobald, M., and Widom, J. LIVE: A lineage-supported versioned DBMS. In *Scientific and Statistical Database Management: 22nd International Conference, Ssdbm 2010, Heidelberg, Germany, June 30-July 2, 2010, Proceedings* (2010), p. 416.

[9] Simmhan, Y. L., Plale, B., and Gannon, D. A survey of data provenance in e-science. *SIGMOD Record 34* (2005), 31–36.

[10] Simmhan, Y. L., Plale, B., and Gannon, D. A survey of data provenance techniques. Tech. rep., 2005.

[11] Stonebraker, M., Becla, J., Dewitt, D., Lim, K., Maier, D., Ratzesberger, O., and Zdonik, S. Requirements for science data bases and SciDB. In *4th Biennial Conference on Innovative Data Systems Research (CIDR09)*.

[12] Widom, J. Trio: A system for integrated management of data, accuracy, and lineage. Citeseer.