Partial Results in Database Systems

Willis Lang, Rimma V. Nehme, Eric Robinson, Jeffrey F. Naughton*
Microsoft Gray Systems Lab, *University of Wisconsin
{wilang,rimman,errobins}@microsoft.com, *naughton@cs.wisc.edu

ABSTRACT

As the size and complexity of analytic data processing systems continue to grow, the effort required to mitigate faults and performance skew has also risen. However, in some environments we have encountered, users prefer to continue query execution even in the presence of failures (e.g., the unavailability of certain data sources), and receive a "partial" answer to their query. We explore ways to characterize and classify these partial results, and describe an analytical framework that allows the system to perform coarse to fine-grained analysis to determine the semantics of a partial result. We propose that if the system is equipped with such a framework, in some cases it is better to return and explain partial results than to attempt to avoid them.

1. INTRODUCTION

In this paper we consider evaluating relational queries over multiple information sources, some of which might return incomplete tuple sets. This situation could arise in a wide variety of scenarios. For example, it could arise with queries spanning a collection of loosely coupled cloud databases, if one or more of them is temporarily down or unusable (say due to network congestion or misconfigurations); we may also see it with queries in a traditional parallel RDBMS, if a node fails during query evaluation and its data becomes unavailable; or it could even appear with queries in a single node system, if some base tables or views are known to be incomplete.

Consider a specific example the authors have encountered: today, with public clouds (e.g., AzureDB), users can sign up for multiple independent instances of relational databases. A significant number of these users choose to "self-shard" (or horizontally partition) their tables across hundreds to thousands of these databases. A critical point is that in such a scenario, each of the sharded relational database systems is an independent entity, and there is no unifying system collectively managing the collection of relational systems. These customers often wish to query over the totality of these systems, but unfortunately, poor latency, connection failures, misconfigurations, or system crashes are all quite possible in any of the loosely coupled databases. At this point the law of large numbers becomes fatal — even with 99.9% uptime, a query over a 1000-shard table will likely have at least one inaccessible shard, and if executing the distributed query requires all of the 1000 systems to be accessible during execution, the query may literally never complete.

In every instance of an incomplete input, our traditional database instincts tell us that the solution is to fix the problem: we should either replicate the data sources comprising the distributed system or make them more reliable; we should add replication and failover to the nodes of our parallel DBMS; or we should embark on data cleaning and repairing efforts to fix the incomplete tables and views in the single node system [4, 5, 7, 10, 14, 16, 20, 30, 33–36]. However, these solutions can be either financially costly, performance hindering, or both. Furthermore, in certain cases, such as querying over loosely coupled cloud sources, an error external to the database or misconfigurations may be impossible to fix. Finally, consistent querying techniques that rely on functional dependencies and integrity constraints currently become inapplicable in this environment. That is why, in this paper we consider a different approach: letting the query run to "completion" despite the incomplete input(s).

In some cases, of course, this is a very bad idea. When reporting numbers to the SEC, or billing a customer, or the like, incomplete answers are not acceptable. However, there are use cases in which the user may be willing to accept an answer computed with incomplete inputs. For example, the user may be doing exploratory work to gain some insight, or may be interested in answering a query like "find me 1000 customers satisfying the following condition..." In such cases, it may be preferable to return imperfect answers rather than to have the query fail, or to incur a delay, or incur the cost and effort of ensuring that such failures happen very rarely.

The astute reader may be wondering "haven't I seen something like this approximate result stuff before?" Undoubtedly the answer is "yes," but here we are working in a very different setting. Rather than viewing query processing as an incremental process in which the query processor systematically explores more and more of the input to yield successively closer approximations to the true result [23, 28, 31], we are interested in query processing in which, due to forces out of the control of the query processor, part of the input is simply not available and will not become available during the query's lifetime.

Of course, merely returning such an answer to an unsuspecting user would be very poor form; the system needs to tell the user that this result is computed based upon incomplete data. Furthermore, the more the system can guarantee about the partial result, or explain to the user about the result, the better. This raises the following intriguing question: is there anything we can say about a computed answer based upon incomplete data other than "be careful, this is based on incomplete data?" For example, can we make any guarantees (about what the user may or may not receive)? Can we classify the types of anomalies that might result and develop sound mechanisms for determining when they can and cannot occur?

SIGMOD'14, June 22–27, 2014, Snowbird, UT, USA.

http://dx.doi.org/10.1145/2588555.2612176 .

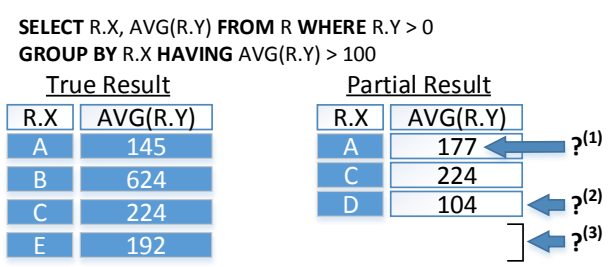

Figure 1: Example: a *True result* versus a *Partial result*.

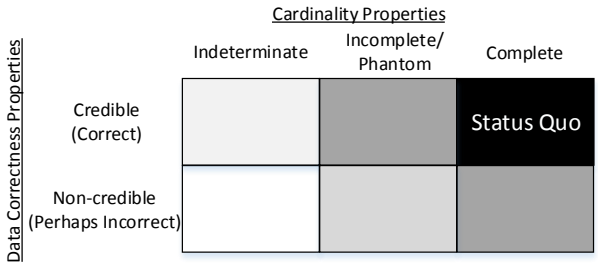

Figure 2: Characterizing partial results.

This brings us to the heart of this work: we present a broad classification of what can "go wrong" when evaluating queries over incomplete data, and show how to detect the various anomalies that arise by analyzing how they are created and propagated through the operators in a query plan. This classification can be used either proactively, where the user specifies "I will only tolerate the following kind of anomalies;" or after the fact, where the system returns "here is what I can tell you about the anomalies that might exist in this result." We present our initial approach to interactively displaying such information to the user in Section 6.

Note that although the root cause of incomplete information may be straightforward, its impact on a query plan is less obvious. The nature of a query result over incomplete inputs is not simply that we receive fewer tuples. Figure 1 shows a simple query that concisely illustrates "all that can go wrong" when a system produces a partial result. In the figure, we consider the result of a simple aggregation query over table R. Suppose that the scan of R is incomplete (perhaps R itself is incomplete, or perhaps R is partitioned and some partition of R resides on a currently inaccessible node.)

In our example, the result for group 'C' is correct, but every other row is problematic. The three main differences between the true result and the result over incomplete data are: (1) the average value calculated for group 'A' is incorrect; (2) a tuple for group 'D' is produced even though it is not found in the true result; and (3) the partial result does not have a tuple for either group 'B' or group 'E'. Each of these anomalies occurred because the scan of table R was not complete, but these anomalies surface at different times and for different reasons during query execution. For anomaly (1), if we are missing any tuples from R that contribute to the group for 'A', it is not hard to see that the average calculation may be wrong. For anomaly (3), the result tuple for group 'B' is perhaps missing, because all of the tuples in R that contribute to group 'B' are missing, while the result tuple for group 'E' may be absent because tuples missing from the scan of R caused the computation of an incorrect average for group 'E', which in turn failed the HAVING clause. Finally, anomaly (2) arose because an incorrect value for the aggregate for group 'D' caused the group to mistakenly satisfy the HAVING clause. Anomaly (2) is unique because it demonstrates that results over incorrect inputs may have "extra" tuples in their output (not in the True result); we call such tuples "phantom" tuples.

This example provides the intuition behind our classification of the errors that can arise when evaluating relational queries over incomplete inputs. Our classification can be viewed as expanding the space of answers produced by database systems as shown in Figure 2. The status quo, of course, is to return only complete and correct tuple sets. But in addition to these complete and correct results, Figure 2 characterizes additional kinds of results with two orthogonal axes: cardinality and correctness. For cardinality we consider four possible categories: complete, missing tuples but no phantom tuples, phantom tuples but no missing tuples, or indeterminate (both missing tuples and phantom tuples). Correctness refers to the values in the tuples that are returned, and is simpler: either the values are credible (correct) or not credible (possibly incorrect). Figure 3 displays this same taxonomy in a different format, representing the options in the cardinality dimension as a partial order.

The challenge, of course, is to determine, for a particular query with a given incomplete input, where in this classification the result set lies. This is the key to returning only the result sets that satisfy user requirements if they impose them. For example, if the user specifies that she wants no phantom tuples, we should analyze the query execution and only return result tuples if we can guarantee phantom tuples are not present, or refuse to return any answer at all if we cannot. It is also the key to providing a user with after the fact information about the result if that is their desire. For example, telling the user that both phantoms and missing tuples could have occurred and letting them decide whether to accept the answer or not. Due to the dynamic nature of failure detection, the classification of a partial result is plan dependent. The precision of our classification depends on the depth to which we are willing to analyze all the information about the missing data and the query. Finally, the last challenge is to communicate our partial result semantics back to the user in a meaningful yet intuitive way.

In this work, we present partial result production as a first-class approach to deal with data access failures in database systems. Existing work that is similar to ours focuses on the basic tenet of producing the true result, albeit through the analysis of iterations of partial answers and through certain operating assumptions [9, 37]. Our work is complementary to existing research in the areas such as online aggregation [23, 28, 31], data provenance and lineage [8, 13, 19, 26, 39], and approximate query processing [2, 11, 15, 18, 32] in that we are providing certainties for an uncertain result *produced due to failures*.

The contributions of this paper are as follows:

- We identify the tuple set properties that can be used to classify partial results. (Section 2)
- We present a partial result analysis framework with four models that determine the degree of our partial result classification precision. (Section 3)
- We describe how relational operators propagate partial result semantics in our framework. (Section 4)
- We describe the relationship between query optimization, failure models, and our partial result classification. (Section 5)
- We implemented a prototype system and discuss how a user may use and tune a partial result-aware system. (Section 6)

Related work follows in Section 7 along with our conclusions and future work in Section 8.

2. PARTIAL RESULTS TAXONOMY

We provide a way to classify results by the guarantees that can be made on different partial result properties. We define a "partial result" as a tuple set that is produced from some query execution where some data needed by the query is unavailable. The system

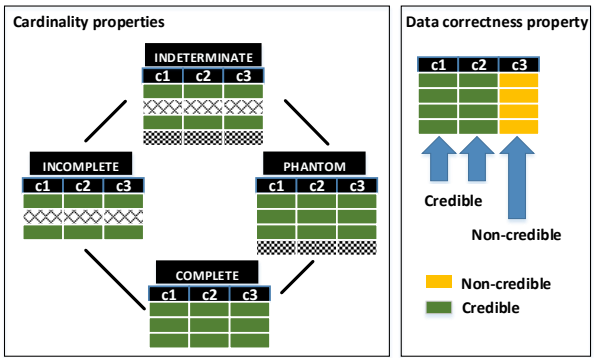

Figure 3: Cardinality and correctness partial results.

has registered the failure and continues query execution using only the available data. Thus, a partial result may not be the same as the result that would have been produced had the query read the base tables completely (the **True result**).

Our partial result semantics are based on a combination of two basic properties of a tuple set. As we show in Figure 3, these two properties are used to describe how a particular partial result tuple set differs from the True result. These two properties are the **cardinality** and the **correctness** of a partial result.

2.1 Cardinality of Partial Results

There are two basic aspects that characterize the **cardinality** of a result set relative to the corresponding True result. The partial result may be missing some tuples, which we call the **Incomplete** aspect; Or, the partial result may have some extra tuples, which we call the **Phantom** aspect.

While it may be obvious how we may classify a result as *Incomplete*, the *Phantom* aspect may not be as straightforward. We have already seen one way that we may encounter the *Phantom* aspect: they can be generated by a predicate over values that are incorrect. This was the case in the "HAVING" clause in Figure 1. Another way Phantom tuples can be produced is by **non-monotone** relational operations such as SET DIFFERENCE. To see this, note that for the SET DIFFERENCE $A - B$, if B is Incomplete, then the result of $A - B$ may have more tuples than if B were complete.

If we cannot rule out the *Incomplete* aspect of the cardinality of a result, and simultaneously cannot rule out the *Phantom* aspect, then we call partial result **Indeterminate**. Conversely, if we can simultaneously rule out both the *Incomplete* and *Phantom* aspects, then we say that the cardinality of the result is **Complete**.

Therefore, given the presence or absence of these two cardinality aspects, a result set's cardinality can be labeled as either *Complete*, *Incomplete*, *Phantom*, or *Indeterminate*. These four labels are placed in a partial order lattice in Figure 3. At the bottom of the lattice, a partial result is **Complete** if we are able to *guarantee* that each of the tuples that are returned correspond to a tuple of the True result. When we *lose cardinality guarantees*, we may "**escalate**" the state of the tuple set to another state. **Escalation** of a partial result (and its properties) means that we have lost the ability to make guarantees that we could make lower in the lattice. As we will see in Section 4, many operators exhibit this escalation behavior under different circumstances.

2.2 Correctness of Partial Results

The other partial result property we consider is the **correctness** of the data values in the result. The cardinality property is separate from the correctness property because *Completeness* does not imply data correctness and vice versa. For example, we can have a partial result tuple set that is guaranteed to be *Complete* even though none of the data values can be guaranteed to be correct. (As a simple example, consider a COUNT aggregation without a GROUP BY clause: we will always get the correct cardinality of exactly one tuple.)

We classify data that cannot be guaranteed to be correct as **Non-credible**, while correct data is classified as **Credible**. For simplicity, in this paper, we assume that the input data read off of persistent (potentially remote) data storage is *Credible* (although this need not be the case in general). This means that data can only lose the *Credible* guarantee when it is calculated (produced by an expression) during query processing. For example, calculating a COUNT over a partial result that is *Indeterminate* means that the result value may be wrong, so we have to classify it as *Non-credible.* We can describe a data set with respect to credibility at different granularities. At the coarsest granularity, we can say the entire result set is *Non-credible*, but sometimes we can do better. For instance, if we know which column was created by an expression evaluation, then we may be able to distinguish some parts of the partial result as *Credible* and other parts as *Non-credible.*

The **correctness** property of a result is not the only property where we can classify a tuple set at different granularities. The **cardinality** property can be further refined for horizontal partitions of the data. Next, we will discuss different analysis levels and data granularities at which we can classify partial results.

3. PARTIAL RESULT ANALYSIS MODELS

We will start by providing an overview of how we can analyze queries to provide partial result semantics. Consider the distributed querying example from Section 1, where we have a "table" defined over a set of remote databases. Suppose a data access failure occurs while querying these databases and the user has elected to accept partial results. Our goal is to provide information to help the user understand the quality of this partial result. Depending on how much we know about what has failed, and how deep we are willing to drill into the semantics of the query, we can provide the user with different partial result guarantees.

Initially, suppose that we know nothing about how the tables are partitioned or the query being executed, and only know that some node that the system tried to access for data was unavailable. In this situation we cannot guarantee any nontrivial partial result properties on the output. This translates to *Indeterminate* and *Non-credible* semantics. However, if we know which query was executed, and which tables were *Incomplete* due to failures, we can make more meaningful guarantees.

Furthermore, if we look into the detailed semantics of the operators applied in the query (e.g., which columns a PROJECT eliminates), we can be even more precise and provide semantics on the vertical partitions of the tuple set. Finally, if we can identify which specific data nodes were unavailable, and we know the horizontal partitioning strategy of the tables, we can classify subsets of tuples (horizontal partitions of the result).

In this section, we will discuss these four models with different analysis "granularities" (see Figure 4). A finer granularity model requires an increased understanding of the data failure that occurred and the operator tree executed. We present four models because this emphasizes that many levels of result quality guarantees are possible. The four models we discuss in this section are representative of a reasonable spectrum of models, and illustrate that there is a tradeoff between the precision and implementation effort required and the guarantees that are possible.

3.1 From Query to Partition Level Analysis

For concreteness, we will use an example based on a TPC-H benchmark Query 15. The view creation query is over a single

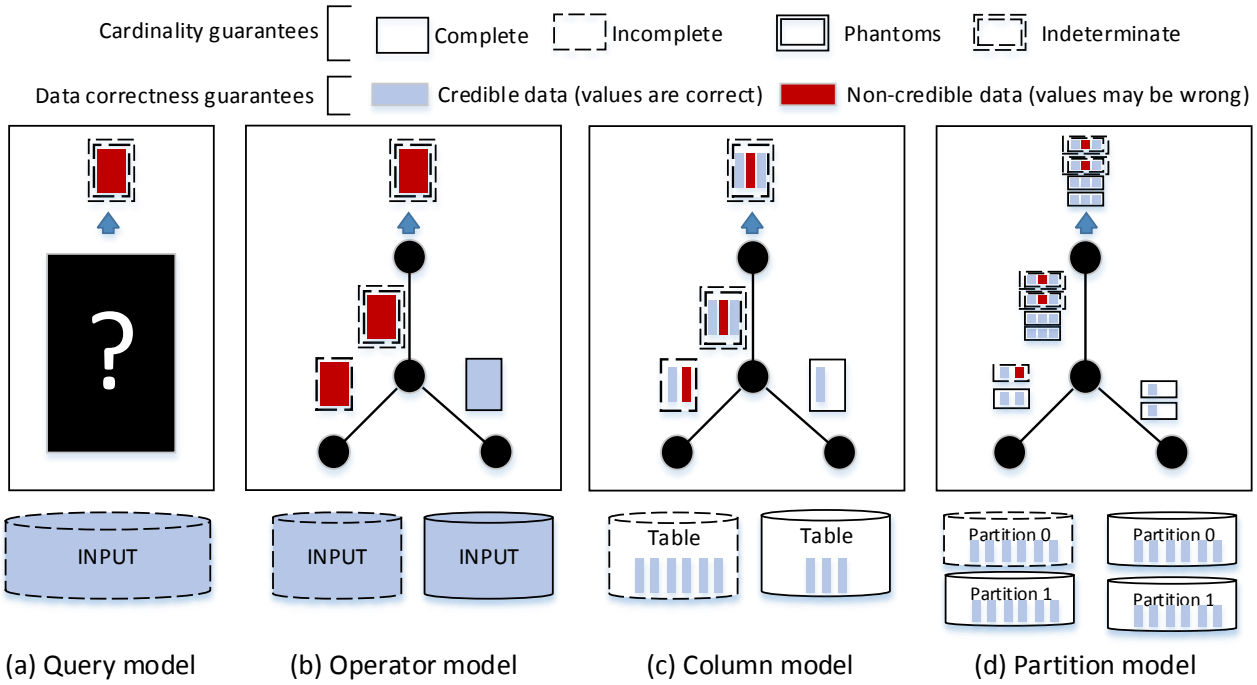

Figure 4: Partial result analysis models.

column name	data type	column name	data type
L_ORDERKEY	identifier	L_RETURNFLAG	fixed text, size 1
L_PARTKEY	identifier	L_LINESTATUS	fixed text, size 1
L_SUPPKEY	identifier	L_SHIPDATE	date
L_LINENUMBER	integer	L_COMMITDATE	date
L_QUANTITY	decimal	L_RECEIPTDATE	date
L_EXTENDEDPRICE	decimal	L_SHIPINSTRUCT	fixed text, size 25
L_DISCOUNT	decimal	L_SHIPMODE	fixed text, size 10
L_TAX	decimal	L_COMMENT	var text, size 44

Table 1: TPC-H LINEITEM table

LINEITEM table (whose schema is shown in Table 1) and the view definition is as follows:

View Definition For TPC-H Q15

```
CREATE VIEW REVENUE (SUPPLIER_NO, TOTAL_REVENUE) AS
SELECT L_SUPPKEY, SUM(L_EXTENDEDPRICE x (1-L_DISCOUNT))
FROM LINEITEM
WHERE L_SHIPDATE >= DATE '[DATE]' AND
L_SHIPDATE < DATE '[DATE]' + INTERVAL '3' MONTH
GROUP BY L_SUPPKEY
```

Consider a few queries over this view. In addition to simply scanning the view, we will consider a query variant that essentially adds a `HAVING` clause to the SUM AGGREGATE:

Q1 – `SELECT * FROM REVENUE`
Q2 – `SELECT * FROM REVENUE WHERE TOTAL_REVENUE > 100000`

Figure 4 describes the four different models of analysis that one can do to determine the partial result semantics when we have a table access failure. We will discuss each of the four models next, starting with the coarsest analysis.

3.1.1 *Query Model*

At the Query Model granularity, we treat a query as a black box that has produced a partial result (Figure 4(a)) given that the input data to it is *Incomplete*. We do not know what is "wrong" with the partial result (i.e., how it deviates from the True result), so we cannot provide any guarantees about it. Therefore, for both queries Q1 and Q2, the partial results that are produced are classified as *Indeterminate* and *Non-credible*:

Query Model:
Q1 – ***Indeterminate, Non-credible***; **Q2** – ***Indeterminate, Non-credible***

3.1.2 *Operator Model*

Now suppose we are willing to look into a query and analyze its logical operator tree. Furthermore, suppose that for multi-table queries, we can distinguish between *Incomplete* and *Complete* tables based on the failures (as we show in Figure 4(b)). With this information, we can now provide stronger guarantees. At this granularity, for each operator in the tree, we need to know the input's partial result semantics (i.e., whether it is *Incomplete*; *Phantom*; or *Credible*). Then, for each operator, we need to determine the semantics of the output tuple set that it returns.

For query **Q1**, we may have the following query plan:

`PROJECT -> SELECT -> SUM`

The input to the PROJECT operator is *Incomplete* (but fully *Credible*) because we were unable to read the LINEITEM table in its entirety.[1] We need to figure out if and how it will change the partial result guarantees of the query's output. Given a tuple set that may be *Incomplete*, but is *Credible*, a PROJECT operator does not change the partial result semantics of the tuple set and simply produces a result labeled with the same semantics as its input.

Moving up the operator tree, the input to the SELECT is still *Incomplete* but *Credible*. Here, the SELECT doesn't change the partial result semantics since all the data is *Credible*, so the output from the SELECT is still *Incomplete* and *Credible*.

Finally, the SUM aggregate takes as input *Incomplete* and *Credible* results and computes a SUM using a single column for the GROUP BY. Given that the input tuple set may be missing some tuples, we cannot guarantee that the SUM produces the correct value. Furthermore, we don't know if we captured all the groups of

[1] In our paper, we consider the SQL version of PROJECT instead of the duplicate-eliminating relational PROJECT operator. Duplicate eliminating PROJECT can be thought of as a composition of PROJECT – AGGREGATE – PROJECT where the AGGREGATE is a simple COUNT with a GROUP BY over all the columns.

Partial Result Credibility Semantics

	→ Query plan operator order →				
Q1 query plan	scan	π	σ	sum	
Q3 query plan	scan	π	σ	sum	σ
L_ORDERKEY	T				
L_PARTKEY	T				
L_SUPPKEY	T	T	T	T	T
L_LINENUMBER	T				
L_QUANTITY	T				
L_EXTENDEDPRICE	T	T	T		
L_DISCOUNT	T	T	T		
L_TAX	T				
L_RETURNFLAG	T				
L_LINESTATUS	T				
L_SHIPDATE	T	T	T		
L_COMMITDATE	T				
L_RECEIPTDATE	T				
L_SHIPINSTRUCT	T				
L_SHIPMODE	T				
L_COMMENT	T				
Σ –TOTAL_REVENUE				F	F

Partial Result Cardinality Semantics

Incomplete	T	T	T	T	T
Phantom	F	F	F	F	T

Table 2: Column-model semantics for Q1 and Q2.

the GROUP BY. Thus the output of the SUM will be labeled with *Incomplete* and *Non-credible* partial result semantics.

Query **Q2** performs a SELECT filter on the aggregated column of the (unmaterialized) view, which essentially can be treated as a `GROUP BY ... HAVING`. Given *Incomplete* and *Non-credible* input, the SELECT "escalates" the partial result semantics to *Indeterminate*. This is because the input values are *Non-credible*, and we don't know if we are correctly allowing tuples to pass the filter or not.

Therefore, the output of these two queries will have the following partial result semantics:

Operator Model:

Q1 – *Incomplete*, *Non-credible*; **Q2 – *Indeterminate*, *Non-credible***

While the operator tree analysis model allows us to distinguish different partial result semantics, it still produces overly conservative guarantees. This is because, while it no longer treats the entire query as a black box, the Operator model still treats the inputs and outputs as black boxes. If we tease apart the columns of a tuple set, it will allow us to be more precise about the partial result semantics, which leads us to the Column model of analysis.

3.1.3 Column Model

At the operator level of analysis, we treat the input and output data as a homogeneous group of data and set the partial result semantics for *all* of the tuples and columns without distinction. If we study partial result semantics at the column-level, then *we are able to discern and track the credibility of different parts of a tuple.* To do this, we must identify the parameters of the operators to know which columns of the tuple they are processing. We now revisit the view definition query from TPC-H Query 15 and show the differences between the Column model of analysis and the prior Operator model analysis.

The operators in the query plan for the view are of course the same PROJECT, SELECT, and AGGREGATE operators considered in the Operator-model analysis. However, each operator is now aware of the credibility of individual columns. In Table 2, we show the column credibility semantics produced by each operator. We can see in Table 2 that for query **Q1**, the columns read off of storage, through the PROJECT, and the SELECT are all *Credible*. The tuple set is also *Incomplete*. However, when we calculate the SUM aggregate over the *Incomplete* tuple set, the resulting TOTAL_REVENUE column is determined to be*Non-credible*. For query **Q2**, the SELECT predicate evaluating a *Non-credible* column (TOTAL_REVENUE) results in an escalation to *Indeterminate* (both *Incomplete* and *Phantom* aspects cannot be ruled out).

LINEITEM partitioning		
	Node 1	Node 2
Column-level	1 <L_SUPPKEY< 10000	
Partition-level	LO: 1 <L_SUPPKEY< 5000	HI: 5001 <L_SUPPKEY < 10000

Table 3: Partition model analysis exploits the systems knowledge of how the table data is partitioned across the data nodes.

The Column model of analyzing partial result semantics provides finer granularity precision for making partial result guarantees:

Column Model:

Q1 – *Incomplete*, *Credible*(L_SUPPKEY) *Non-credible*(TOTAL_REVENUE)

Q2 – *Indeterminate*, *Credible*(L_SUPPKEY) *Non-credible*(TOTAL_REVENUE)

Compared to the partial result semantics produced when using the Operator model, we now know that certain columns of the output have correct values. For the two queries, we now have a mix of *Credible* and *Non-credible* columns, which can be considered the hallmark of the Column model of analysis (as shown in Figure 4(c)).

3.1.4 Partition Model

So far we have only considered what happens when the entire input data is classified as *Incomplete* or *Complete*. In the Partition model, by contrast, we consider the input table to be a collection of partitions, and use properties of partitions in our analysis. In large-scale parallel data processing systems typically data is partitioned according to appropriate partitioning schemes [40].

Recall our example working environment of querying over loosely coupled remote databases, where a table is "sharded" across the individual shards. If we know which nodes were unavailable or returned incomplete data, then we can classify the other partitions of the table as *Complete* and *Credible*. This means that, if the partition properties can be propagated through the analysis of the query, we can determine that certain partitions of the result match the corresponding partitions in the True result. This is depicted in Figure 4(d), where Partition-level analysis breaks all of the tuple sets (input, intermediate, and final) horizontally into partitions. If we revisit our running example of querying over TPC-H Q15's view, then we will find that the partition-model analysis gives us an even more precise classification than column-model analysis.

Assume that the LINEITEM table was partitioned across two nodes using the L_SUPPKEY column. Call one partition 'HI' and the other 'LO', where the HI partition has the half of the tuples with the larger L_SUPPKEY values (see Table 3). The input to our queries **Q1** and **Q2** are now the two partitions of LINEITEM where one is *Complete* (e.g., HI) and the other is *Incomplete* (LO).

When the initial PROJECT operator takes the tuples from the *Complete* partition (HI) as input, it produces a *Complete* (and still fully *Credible*) output. On the other hand, when it processes the *Incomplete* partition, the output analysis is the same as the Column-level analysis: *Incomplete* and all columns are *Credible*. Here, the PROJECT processes these two partitions and the output can be divided into two partitions because the partitioning column, L_SUPPKEY, was retained. Next, the SELECT operator processes the two partitions in the same manner as the PROJECT. Its output can also be thought of as two separate partitions: the HI tuples and the LO tuples. Again, it is key that the SELECT operator does not remove columns, so we retain the partitioning knowledge in L_SUPPKEY. Finally, since the SUM operator performs a GROUP

Partial result semantics	Query model		Operator model		Column model			Partition model			
	card.	cred.	card.	cred.	card.	cred.		part	card.	cred.	
						L_SUPPKEY	TOTAL_REVENUE			L_SUPPKEY	TOTAL_REVENUE
Q1: SELECT L_SUPPKEY, TOTAL_REVENUE FROM REVENUE	Indet.	F	Incomp.	F	Incomp.	T	F	HI	Complete	T	T
								LO	Incomp.	T	F
Q2: SELECT L_SUPPKEY, TOTAL_REVENUE FROM REVENUE WHERE TOTAL_REVENUE > 100000	Indet.	F	Indet.	F	Indet.	T	F	HI	Complete	T	T
								LO	Indet.	T	F

Table 4: Side-by-side comparison of partial result semantics determined using four different analysis models for Q1 and Q2.

BY on L_SUPPKEY, its output tuples are also partitioned into the HI and LO partitions. Here we see the advantages of Partition-level analysis. Since the HI partition was *Complete* and all the columns were *Credible*, the SUM on any of the HI groups is correct and can be classified *Credible*. This means the partial results of query **Q1** will have semantics as follows:

Partition Model:

Q1:

HI – {*Complete*, *Credible* (L_SUPPKEY, TOTAL_REVENUE)}

LO – {*Incomplete*, *Credible* (L_SUPPKEY) *Non-credible* (TOTAL_REVENUE)}

Since query **Q2** essentially adds a SELECT operator to process the results of the AGGREGATE, it will also take the HI and LO partitions as input. The partial result semantics of **Q2** is:

Q2:

HI – {*Complete*, *Credible* (L_SUPPKEY, TOTAL_REVENUE)}

LO – {*Indeterminate*, *Credible*(L_SUPPKEY) *Non-credible*(TOTAL_REVENUE)}

We can see that with partition-level analysis, for all partial results, we are able to identify and return some tuples that are exactly the same as in the True result. The partition model for analysis provides the most precise guarantees in its partial result semantics by providing the finest granularity in its data classification. However, it is also the most complex.

Summary: In Table 4, we summarize our analysis for two queries based on the TPC-H Q15 view as we draw tighter boxes around the data. The partial result semantics for each query using the four levels of analysis are shown with the coarsest Query-level granularity on the left and the finest-granularity Partition-level analysis on the right. As we move from left to right, we see that we are able to classify more and more of the result set as *Complete* and *Credible* which provides better value for the user.

4. PROPAGATION OF PARTIAL RESULT SEMANTICS

In the previous section we illustrated by examples how the relational operators of a query may change the partial result semantics of a tuple set as it processes the data. In this section, we will select a few common relational operators (four unary operators and three binary operators), and describe in detail their behavior with respect to partial result semantics.

Here we discuss the way operators propagate partial result semantics using the Partition model of analysis. Since the other three models are essentially "roll-ups" of the Partition model in terms of precision, the operators' behavior in those models can all be derived from our description of the Partition model.

4.1 Unary Operators

The four unary operators we study are SELECT, PROJECT, EXTENDED PROJECT, and AGGREGATION. For the SELECT operator, we will limit our scope to "simple" predicate types that involve expressions (using $<, \leq, =, \geq, >, <>$) on the columns of the tuples being processed. We differentiate projection into two categories: those that simply remove columns (PROJECT), and those

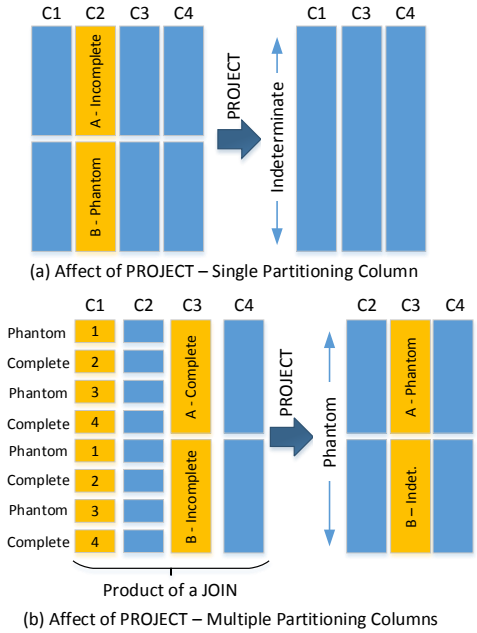

Figure 5: (a) Removing the sole partitioning column. (b) Removing a partitioning column from a tuple set that has multiple partitioning columns.

that can define a new column through an expression (EXTENDED PROJECT). For the AGGREGATE operators, we only consider the basic types: COUNT, SUM, AVG, MIN, and MAX.

For each operator we will describe how it is affected by the input with certain partial result semantics and how it defines the partial result semantics of the result set.

4.1.1 SELECT

As we discussed in Section 3, the SELECT operator affects partial result semantics if it has a predicate expression that operates over columns that are *Non-credible*. In that case, since we don't trust the data values that we are evaluating expressions over, then we can't be confident of the elimination of tuples and the retention of tuples. In this case we must set the cardinality property of the result to *Indeterminate*. If the predicate is defined over all-*Credible* data, it simply propagates the partial result semantics from input to output.

4.1.2 PROJECT

The PROJECT operator only affects the partial result cardinality property of a tuple set. The only way it can do this is when the tuple set is partitioned. Figure 5 illustrates how the PROJECT operator can "**taint**" the semantics of a tuple set if it eliminates the partitioning column. Figure 5(a) shows a simple partitioned tuple set where partition 'A' is *Incomplete* and partition 'B' is *Phantom*. If the PROJECT operator eliminates the partitioning column C2, then the tuple set becomes a single "partition" and we can no longer know if tuples are missing or if *Phantom* tuples exist (thus making

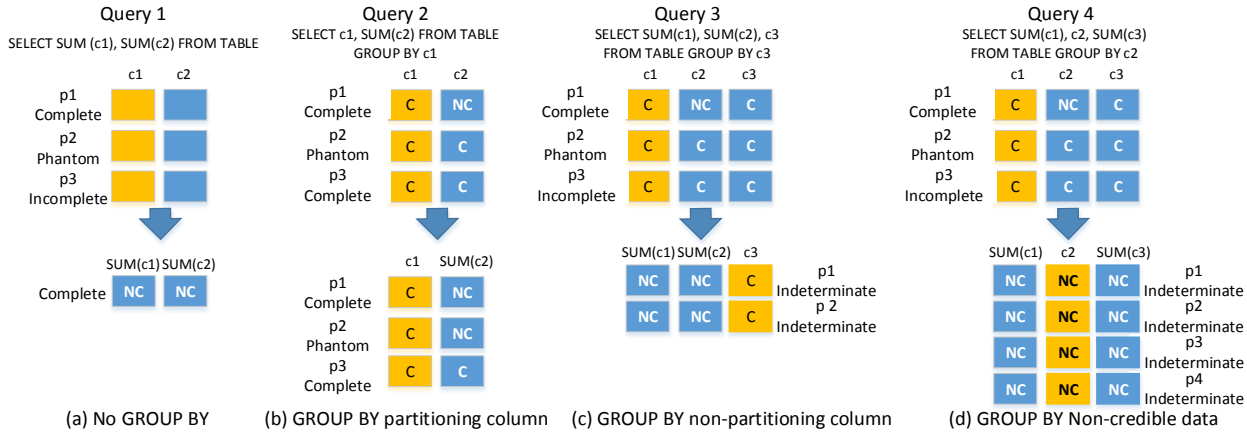

Figure 6: Aggregation operators behave very differently depending on which columns are used in the GROUP BY clause. C – *Credible*, NC – *Non-credible*, the partitioning column is shaded lighter.

it *Indeterminate*). Hence the resulting "merge" of the two partitions taints the result set.

In Figure 5(b) we look at a more complicated example, where the input to the PROJECT operator is the product of a CARTESIAN PRODUCT/JOIN (see Section 4.2.2). Column C1 is 4-way partitioned while column C3 is 2-way partitioned. Column C1 has two partitions (partitions 1 and 3) that have *Phantom* semantics while the other two are *Complete*. Column C3's partitions are *Complete* and *Incomplete*. If we remove C1 through PROJECT, we can no longer identify sets of tuples that may harbor *Phantom*s. Therefore, both partitions defined by C3 are now tainted by the partitions of C1 and thus are escalated to *Phantom* and *Indeterminate*.

On the other hand, if the PROJECT operator removes a non-partitioning column, then PROJECT simply propagates the remaining rows' partial result semantics. Intuitively, in this case the PROJECT operator is not affected by, nor does it affect, the credibility of columns.

4.1.3 EXTENDED PROJECT

The EXTENDED PROJECT operator can create a new column using an expression that may rely on the other columns of the tuple set, and so it is affected by input data with Non-credible columns. Intuitively, if an expression computes a value using *Non-credible* values as input, then the output is also *Non-credible*. If the expression parameters are all *Credible*, then this operator produces a column that we can guarantee as *Credible*. The EXTENDED PROJECT operator does not affect the cardinality semantics (i.e., *Incomplete* and *Phantom*) of a partial result.

4.1.4 AGGREGATE

We consider five types of AGGREGATE functions in this operator: COUNT, SUM, AVG, MIN and MAX. To simplify our discussion, we only consider instances where we apply the functions over one column of the input tuple set.[2] We also assume that there is no implicit PROJECT operation happening over the input that is eliminating columns (i.e., if five columns are provided as input, the output will also have five columns).

The four queries we use to describe AGGREGATE behavior are shown in Figure 6. In the figure, C means the data is *Credible*, NC means the data is *Non-credible*, and the partitioning column is lightly shaded. First, Figure 6(a) (Query 1) shows that an AGGREGATE without any GROUP BY clause always creates a single tuple, so it will always be *Complete*. AGGREGATE is the only operator with this ability to take a non-Complete (Phantom, Incomplete, or Indeterminate) input and produce a *Complete* output. However, we show that if any of the input partitions are not *Complete*, the results will always be *Non-credible*.

In Figure 6(b) (Query 2), if the GROUP BY clause is over partition columns, then the output rows will take the partial result semantics of the source partitions. For example, if a partition has *Phantom* semantics, then the resulting tuple is also classified as *Phantom*. Furthermore, similar to Query 1, computing over Non-complete data results in *Non-credible* results. (Query 4 shows what happens if we GROUP BY a column that is *Non-credible*.)

Figure 6(c) (Query 3) shows a GROUP BY over a non-partitioning column that is *Credible*. In this case, since the input had *Phantom* and *Incomplete* partitions, the output tuples are tainted by these partitions and are escalated to *Indeterminate* (see Figure 3).

Finally, in Figure 6(d) (Query 4), we show that if the GROUP BY clause includes any columns that have *Non-credible* values, then all the output is escalated to *Indeterminate* and all data is deemed *Non-credible*. This is the only way a partitioning column becomes *Non-credible*, since by our assumption, partitioning columns read from the base tables are deemed *Credible*.

4.2 Binary Operators

The binary operators we consider are UNION ALL, CARTESIAN PRODUCT, and SET DIFFERENCE. Using Figure 7, we will describe examples that illustrate the key partial result behavior of these three operators.

4.2.1 UNION ALL

The UNION ALL operator takes two tuple sets and creates a new one by combining all of the tuples. UNION ALL's partial result behavior is to escalate the cardinality property of the output based on the combination of the two input's cardinality properties. For data correctness, an output column is escalated to *Non-credible* if either of the corresponding input columns are *Non-credible*.

In Figure 7, we show two examples that illustrate this behavior. In Figure 7(a), two tuple sets with identical partitioning strategies are given as input. The output will maintain this partitioning strategy, and thus the semantics of partition 1 becomes *Indeterminate* because the two inputs were *Phantom* and *Incomplete*. Furthermore, the credibility of column 2 is lost because one of the inputs for column 2 was *Non-credible*. Similarly, for partition 2, the escalation (based on the lattice "join" in Figure 3) of *Incomplete* and *Complete* results in an *Incomplete* result.

[2] Any scenario where we want to compute an aggregate over an expression can be broken down into an EXTENDED PROJECT followed by an AGGREGATE.

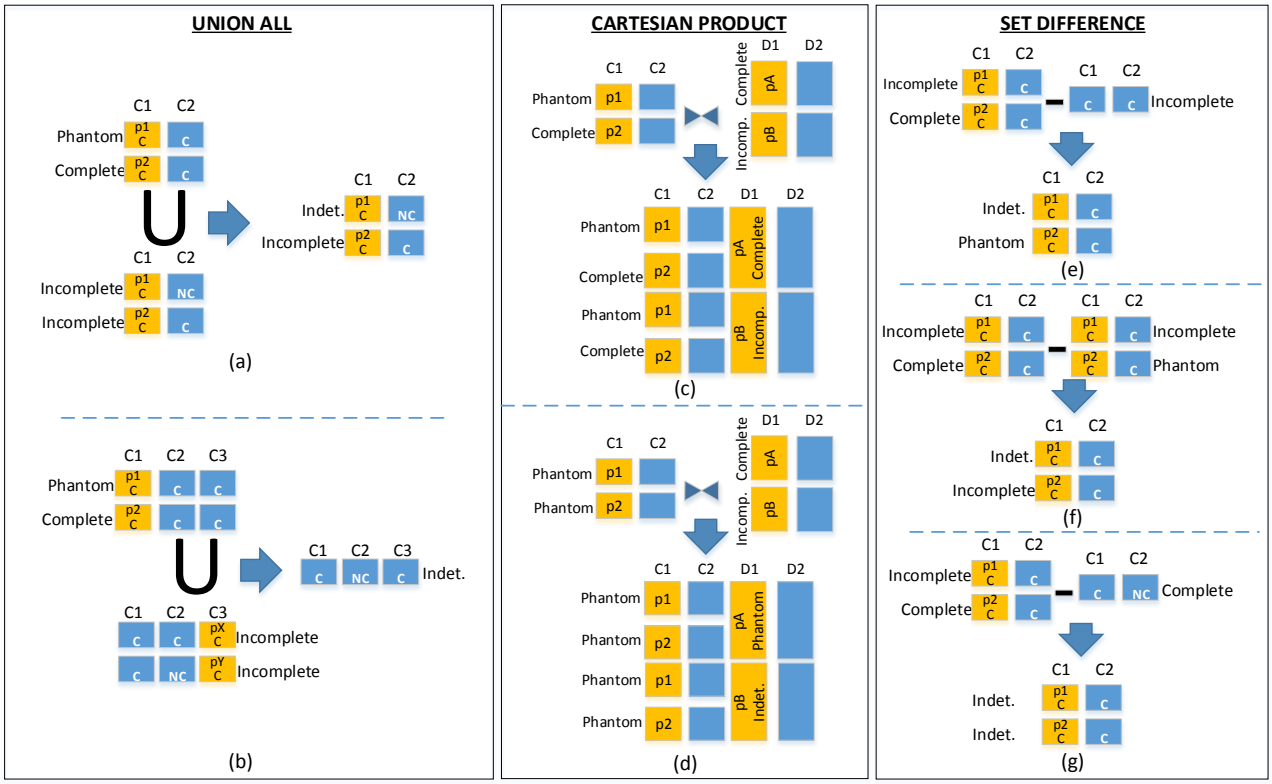

Figure 7: Examples of how UNION ALL, CARTESIAN PRODUCT, and SET DIFFERENCE behave when given partial result input. C – *Credible*, NC – *Non-credible*, the partitioning column is shaded lighter and partitions are identified by "p<ID>".

In the second example in Figure 7(b), we show that if we don't have partition alignment for the two inputs to UNION ALL, we lose all the partitions. The result is then considered a single "partition," where all of the cardinality semantics of the input partitions escalate the output. In this case, the result is *Indeterminate*. The column credibility is also escalated to *Indeterminate* for column C2.

4.2.2 CARTESIAN PRODUCT

The CARTESIAN PRODUCT is relatively straightforward in its behavior. It performs a cross of the two sets of partitions to create the output. It is not affected by, nor does it change the credibility of the data values. However, the CARTESIAN PRODUCT may or may not simply propagate the input semantics to the output. In Figure 7(c), we show a case where the semantics of the input partitions are all retained.

In contrast, in Figure 7(d), we show how the CARTESIAN PRODUCT operator can cause partial result semantic tainting. In Figure 7(d), all of the partitions of C1 were (homogeneously) *Phantom*, the CARTESIAN PRODUCT taints the cardinality semantics of partition A (column D1) to *Phantom* and partition B to *Indeterminate*.

4.2.3 SET DIFFERENCE

In Figure 7(e)-(g), we depict three scenarios for the SET DIFFERENCE operator. As we have mentioned, SET DIFFERENCE is a non-monotone operator so it can create *Phantom* results. In the first example in Figure 7(e), we see that *Phantom* semantics are set if the second input is *Incomplete*. Here, the second input is not partitioned, thus both input partitions get tainted by the *Phantom* aspect.

The second example (Figure 7(f)) shows what happens if the second input is partitioned. In this case, the corresponding partitions are processed together: p1 with p1 and p2 with p2. We also show that if the second input has *Phantom* semantics, the output is tainted by the *Incomplete* aspect since we may have removed tuples we shouldn't have.

Finally, in Figure 7(g), we show that if the second input has *Non-credible* data, all partitions of the result are escalated to *Indeterminate* since we cannot trust the presence (or the absence) of any tuples in the output. If the first input has any *Non-credible* data, then we would also escalate the partial result to *Indeterminate*.

5. DEPENDENCE ON QUERY PLAN

At an abstract level, we envision our classification system being applied as follows. The optimizer chooses a plan to be run; the system then begins running the plan, which consists of a tree of operators. The data accessed by these operators is stored in multiple shards. If at any point during execution an input to an operator "fails" (perhaps the site is or becomes unavailable), then we use the techniques from Section 4 to determine and propagate the effect of these errors up the query plan. Each operator in turn passes the result of this analysis to operators further up in the tree, until at the root, the answer set is classified.

5.1 Dynamically Detecting Failures

Since errors are determined dynamically and by the specific plan executed, it is reasonable to ask how the result classification depends upon the plan chosen. After all, a foundational principle of query evaluation in traditional settings is that the same result is computed independent of the plan; it would be nice if this carried over to partial results analysis so that here, the result classification was independent of the plan. Unfortunately, this is not the case when we consider failures during execution, for at least two main reasons, neither of which are due to our analysis or propagation models.

First, consider two plans (L1) $R \bowtie S$ and (L2) $S \bowtie R$ where the join is computed by a hash-join operator. Here L1 and L2 differ

in that they reverse the build and probe relations of the hashjoin. Now suppose that it turns out that some shard storing a partition of R fails during the execution. The question is when. If the shard fails during the later part of the execution, it is possible that plan L1 may not even "see" this, since it may have completed its read of R before the failure, whereas plan L2 might see the failure, if it occurred during the scan of R at the end of the query plan.

Here we have a remarkable result that the query result itself differs depending upon which plan is chosen. We emphasize that this is not the "fault" of any design decision; we think it is actually reasonable in the world of unplanned failures in large distributed computations. However, it definitely means that our result classification is not independent of the plan chosen; it would seem unreasonable to expect it to be.

This does raise another question: what about scenarios where the failures do not impact the final result? Is it possible that, whenever two plans give the same result in an execution possibly containing failures, our classification scheme always yields the same classification? Unfortunately, the answer is again "no." Consider two physical plans P1 and P2 for a simple selection query on a relation sharded across multiple loosely coupled data sources. Plan P1 scans all of the data sources in parallel applying the selection. Plan P2 is more clever, using a global index that matches the selection predicate, and thus it is able to execute the query by only consulting the subset of shards that actually contain results to the query. The alert reader will likely see what is coming: suppose that some node(s) that contains no results has failed. Plan P1 will see the failure, but Plan P2 will not, because it does not even access the failed node(s).

Of course, this dependency on plan choice occurs even in traditional centralized systems — as a contrived example, one can imagine a situation where a table has a corrupted index, so the plans that use the index will fail while the plans that don't will succeed. What is new here is that we are now accepting partial query results and trying to classify their properties — this exposes the interaction between plans and failures.

At this point the reader might wonder if there are any guarantees we can make whatsoever. It turns out that this is tied to the class of plans and failures considered. To illustrate this, the rest of this section will focus on the following: first, we consider the case where all failures occur *before* the query begins executing and persist throughout the entire execution (we call this the "**persistent failure model**"); second, we consider plans that are equivalent modulo transformations enabled by exploiting the relational algebraic property commutativity. As the following section shows, under these assumptions, we can say that equivalent plans yield identical partial results classification.

5.2 Partial Results Consistency with Operator Re-orderings

In this section we will show the following: *under the persistent failure model, for different orderings (plans) of commutative operators, we will have identical classifications of the partial result outputs.*

Problem Statement: For two commutative operators α and β, we wish to show that for the sequences $(\alpha \rightarrow \beta)$ and $(\beta \rightarrow \alpha)$, under the persistent failure model, the partial result guarantees produced by our analysis are identical for any (partial) inputs.

The persistent failure assumption means that for any set of reorderings, the (partial result) input to the operator plans will be the same, and also, no failures occur in the middle of the plans. Due to space limitations, we do not provide a formal proof. However, we will thoroughly illustrate how all pairs of commutative operators from our operator set maintain partial result consistency when reordered.

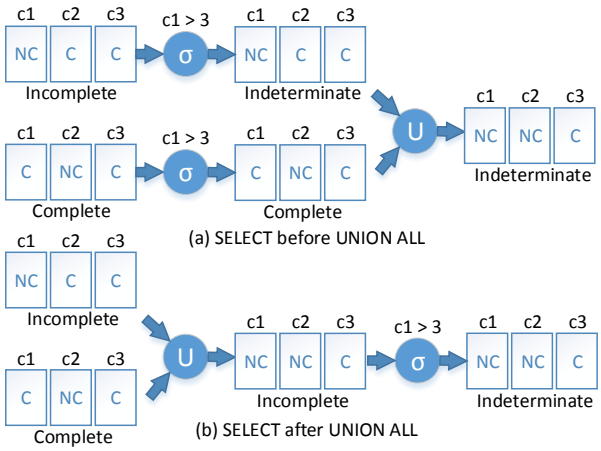

Figure 8: Commutative pairs of SELECT and UNION ALL.

5.2.1 *Commutative Pairs of Unary Operators*

Commutative pairs of unary operators are two operators α and β, such that given an input tuple set R, if we execute either α–> β or β–>α, then the output will always be S . This restricts the pairs we need to examine because of the conditions under which two operators are commutative. For example, a PROJECT cannot eliminate the column that a SELECT requires for its predicate.

Below we will discuss commutative pairs where one of the operators is a SELECT; and a commutative pair of PROJECT and EXTENDED PROJECT. AGGREGATE is not commutative with PROJECT or EXTENDED PROJECT (recall we define AGGREGATE to use all of the tuple's columns for either GROUP BY or the aggregate function.)

Pairs involving SELECT: We want to show that for any pair involving a SELECT, given any ordering, the classification of the output will be the same. The key behavior of SELECT is that it only affects partial result properties if the predicate is on a *Non-credible* column. The propagation behavior of SELECT (when the predicate is defined on all-*Credible* columns) means: *if the SELECT predicate is defined over solely Credible columns, then any commutative pairing with any operator will have consistent partial result semantics.*

Now we only need to discuss when the SELECT predicate is defined over *Non-credible* columns. In this case, the SELECT operator doesn't change any of the credibility properties, but escalates the output cardinality to *Indeterminate*. Therefore, it suffices to show that if we pair the SELECT (with a *Non-credible* predicate) with any of the other three unary operators, the output will always be *Indeterminate* and the credibility of the partial result will be consistent. We can reason about this by "proof by contradiction" reasoning. Suppose we paired a SELECT (with a *Non-credible* predicate) with either PROJECT, EXTENDED PROJECT, or AGGREGATE whereby (a) the result was not classified as *Indeterminate* in the ordering where the SELECT operator is first, and/or (b) different operator orderings result in data being classified as *Credible* and *Non-credible*. Then case (a) is not possible because the only operator that can "de-escalate" the *cardinality* property is AGGREGATE (when there is no GROUP BY) and here, GROUP BY columns must exist for SELECT to be commutative with AGGREGATE. Case (b) is not possible because under the commutative conditions and the *Non-credible* SELECT predicate, the AGGREGATE will only produce *Non-credible* data, and EXTENDED PROJECT is unaffected by the cardinality property.

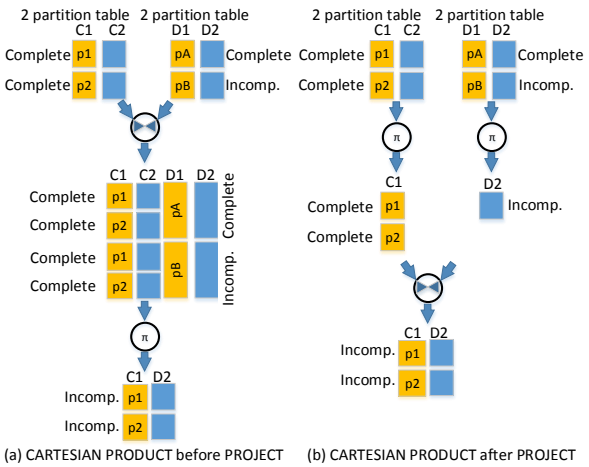

Figure 9: Commutative pairs of PROJECT and CARTESIAN PRODUCT (light shade – partitioning columns).

Commutative PROJECT and EXTENDED PROJECT: Since the PROJECT operator's behavior is only concerned with the partial result cardinality property while the EXTENDED PROJECT operator's behavior is only concerned with the partial result data correctness property, we will always classify the result the same way regardless of the ordering of this pair.

5.2.2 *Commutative Unary/Binary Pairs*

We will break up this discussion into three parts for three different unary operators. The AGGREGATE operator is not generally commutative with our binary operators so we omit it.

Pairs involving SELECT: If the SELECT predicate is defined over all-*Credible* data, the operator simply propagates the partial result semantics of its input to its output. Therefore, similar to the argument for unary operator pairs with SELECT, for any unary/binary pair where the SELECT predicate is defined over all-*Credible* data, partial result semantics are consistent in the presence of re-ordering.

If the SELECT predicate is defined over a column that is *Non-credible*, the SELECT always escalates the result's cardinality property to *Indeterminate*. Furthermore, we know that the SELECT operator does not affect the column's credibility. The UNION ALL operator is commutative with the SELECT operator. UNION ALL "taints" (see Section 4.1.2) its output's partial result semantics with the semantics of its inputs. As we show in Figure 8, the SELECT, UNION ALL commutative pair maintains partial result consistency because the SELECT will always receive the *Non-credible* column for its predicate.

Pairing SELECT with either the CARTESIAN PRODUCT or the SET DIFFERENCE operator does not result in an inconsistent partial result classification. Using the similar reasoning by counter example that we used above, there is no condition under which an operator re-ordering leads to different partial result semantics.

PROJECT: The PROJECT operator is only commutative with the UNION ALL and CARTESIAN PRODUCT operators. For PROJECT and CARTESIAN PRODUCT, we show in Figure 9 that regardless of how we order the two operators, the PROJECT operator will always taint the output of the pair in the same way. This is because the CARTESIAN PRODUCT "crosses" all of the partitions from each input and combines the inputs' columns together. In Figure 9(a), the CARTESIAN PRODUCT operator propagates the input partitions' semantics to its output and then the PROJECT operator "taints" its output when it removes the partitioning column D1. In Figure 9(b), the PROJECT removes the partitioning column D1, creating a single *Incomplete* partition. Then, the CARTESIAN

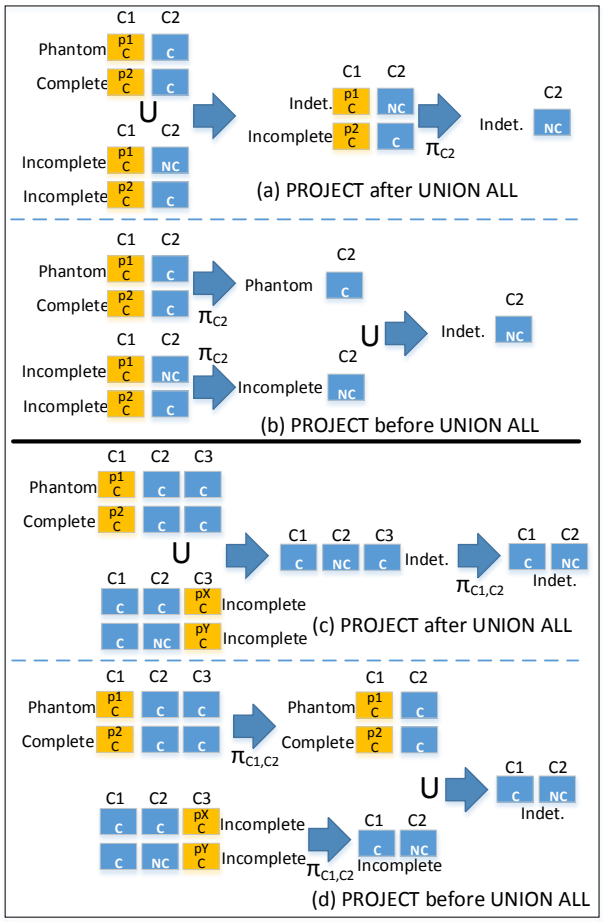

Figure 10: Commutative pairs of PROJECT and UNION ALL (light shade – partitioning columns).

PRODUCT operator taints its output partitions' semantics, since one of its inputs is homogeneously *Incomplete*. As a result, both orderings result in two partitions classified as *Incomplete*. Neither of these operators are affected by or can affect the data credibility property. Thus, PROJECT and CARTESIAN PRODUCT commutative pair have consistent partial result semantics.

For the PROJECT and UNION ALL pair, we only need to consider the case where PROJECT removes a partitioning column (since removal of a non-partitioning column is straightforward). Furthermore, we only need to consider the partial result cardinality classification since the PROJECT operator's behavior does not affect credibility. In Figure 10, two examples illustrate that this pair of operators will be classified consistently regardless of the order. In Figures 10(a) and (b) we show the case when the two inputs have partition alignment (C1), while Figures 10(c) and (d) show the case where the two inputs don't have partition alignment (C1 and C3).

EXTENDED PROJECT: Commutative pairs of EXTENDED PROJECT – UNION ALL operators or EXTENDED PROJECT – CARTESIAN PRODUCT operators have consistent partial result semantics. Since EXTENDED PROJECT's (XPROJ) behavior only involves data credibility and CARTESIAN PRODUCT is concerned with partial result cardinality, it is straightforward to see that we will always classify the result consistently.

Similarly, since XPROJ does not affect the cardinality semantics of a result, for pairs with the UNION ALL operator, we only need to show that a new column produced by the XPROJ operator that is *Credible* in one ordering cannot be *Non-credible* in the opposite operator ordering. The reason this is true is straightforward to un-

derstand: if the new column were ever *Non-credible* in one ordering, then this would mean that one of the inputs to the pair of operators had a *Non-credible* column necessary for the XPROJ expression. Therefore, since the UNION ALL operator taints the output's credibility with both of its input's credibility, we will always classify the result consistently.

5.2.3 *Pairs of Binary Operators*

Binary operators are only commutative (and associative) with themselves (i.e., we cannot have a heterogeneous set of binary operators that is commutative.) Thus, we only need to show that pairs of homogeneous binary operators will be consistently classified. Due to space limitations we will omit this discussion in this paper.

6. EXPERIENCE & IMPLEMENTATION

So far, we have discussed the way a database system can analyze queries to produce partial result guarantees in the presence of input failures. Of course, another important aspect of partial results is how users can control and use a partial result-aware system along with the impact of implementing such a framework into a system.

First, we will discuss how users may interact with partial result-aware database systems. We consider two aspects of user interaction: user input to the system, and presentation of the partial result output to users. Both aspects are key to increasing the value of partial results to a user. We will briefly describe our initial approach to how users may interact with a partial result-aware system. Clearly other approaches could be considered as well, and we present the ideas in this section as a high-level example of what could be done, rather than as a definitive proposal for the best way to interact with partial result-aware systems.

A user that elects to receive partial results from a database can control how the database behaves to ultimately increase the value of a potential partial result output. For example, depending on whether or not the consumer of the result is a human or an application, the user may wish to receive *any partial result* or may chose to *set constraints* that limit the types of anomalies that are acceptable. In the former case, perhaps a human is doing exploratory, ad-hoc data analysis and is willing to accept any result anomaly. In the latter case, an application may accept only certain partial result classifications such as *Incomplete* and *Credible* results, otherwise return an error. In all of these cases, we may wish to provide the user with a way to signal her intentions to the system; perhaps in the form of session controls, DDLs, or query (or table) hints.

On the output side of the problem, there may be many different ways that a partial result can be presented to the user. If we consider the ad-hoc, exploratory user who accepts all partial results, perhaps an operator-by-operator style presentation of how the partial result guarantees are made would be useful. In Figure 11, we show a prototype interface that we have built and connected to our real, partial result-aware database system. Here a user can zoom in on any operator of a query plan and examine the partial result guarantees made about the data at that point. In the figure, the focus is on the PROJECT operator before a CARTESIAN PRODUCT. With this style of interface, the user may wish to receive the partial result output from any operator in the plan to maximize the value of the query's execution. Alternatively, one can imagine an interface that presents the actual raw data output to the user with appropriate meta-data tags. Perhaps with these interfaces, a user may even wish to "bless" the result at a given point in the plan to manipulate the meta-data tags directly and gauge the effects. While our paper has focused on how a database system can internalize the notion of partial results, ultimately, the way users can interact with the system is key to providing value through partial results.

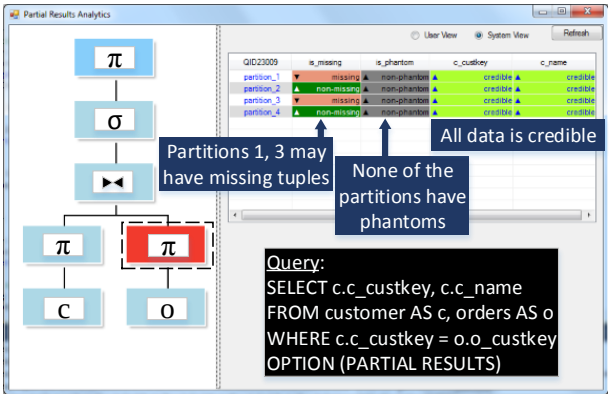

Figure 11: An interface example for partial results exploration.

Incorporating partial results analysis into an existing DBMS requires minimal changes to the codebase, and has almost no impact on the performance of the system. When failures occur, they must be detected, which most systems already do today. Instead of returning an error message when some base data is unavailable, the engine continues the query execution (as if it had 100% of the data), and just before the final answers are returned back to the user, the partial results analyzer (a stand-alone component) would take the detected runtime failures and the query plan used as its inputs and produce the partial results guarantees. This implementation whereby failures are simply passed to the stand-alone analyzer at the end also allow us to manage intermediate data access errors. We would simply retag the inputs or outputs of certain operators if some failure happened between any two operators. Our framework does not impose anything that precludes intermediate failures from being detected and applied. These guarantees along with the result tuples are returned back to the user as the final answer.

7. RELATED WORK

Our paper is related to many research areas, including fault tolerance in big-data systems, data provenance and lineage, online aggregation, approximate query processing, and data quality. Perhaps the most closely related works to ours include the "partial answers" work by Bonnet et al. [9], the view lineage paper by Cui et al., the Trio project [17, 39], and the semantic analysis of approximate answers by Vrbsky et. al., with their APPROXIMATE system [37]. While Bonnet et al. focused on combining partial answers into a complete answer (using "parachute" queries), their work does not focus on classifying and analyzing partial answers and how the operators in the query pipelines may propagate the partial results semantics [9].

The Trio project focuses on a data model, query language, and execution strategies for a system that allows the querying of either inexact data or lineage data [3,17,39]. Our work here is different because we are describing how to identify and derive characterizations for partial results when failures occur while querying traditional relational data. The semantic tags produced by our framework along with the partial results could be stored within the Trio system.

The APPROXIMATE system focused on monotone query processing and did not study aggregations or data credibility (correctness of values) issues [37]. Other related works includes characterizing errors in cardinality estimates [25,27] and identification of phantoms through non-monotone operators [38], which cover only certain aspects of the entire "partial result" space that we study here.

In the theoretical database research community, "naïve querying" over incomplete databases discussed query evaluation and "certain answers" over relations that have null values due to causes such as incomplete data integrations [21]. Other theoretical work includes

the seminal work that characterizes consistent data in an inconsistent database for the purposes of database repair [5]. Such work relies on the knowledge provided by the semantics of integrity constraints and functional dependencies whereas in our target environment of loosely coupled, independent cloud databases, these logical semantics are generally not available. If future cloud services are able to capture user-provided (or even machine-learned) constraints and dependencies (and monitor and enforce them) across 100s to 1000s of cloud databases, then we may be able to leverage the complex reasoning techniques of these prior works to identify tuples that we may have lost or tuples that could never have been present. This is the focus of our future work.

Computing partial results in the absence of complete table data is also similar in spirit to producing results via online aggregation [23, 28, 31], continuous querying of data streams [6, 12], and querying over sampled data [2, 22, 24]. These systems heavily rely on random sampling to provide an "early returns" approximation of a correct answer that may otherwise take much longer to compute. We, on the other hand, attempt to provide guarantees with respect to which parts of the returned results are identical to those in the true result, and which ones are not, and how they may differ if they do indeed differ. Ultimately, our approach provides a taxonomy of partial results that could be utilized by all of these other systems to further subdivide results produced based on their types and provide even tighter guarantees.

The research problem of data lineage and data provenance that provide the capability to see and track back where the data came from has been well studied in many contexts including data exploration, privacy and security, and uncertain database systems [1, 8, 11, 13, 15, 18, 19, 26, 29, 32, 39, 41]. While those approaches focused on developing logical data models, query languages, and physical execution methods for querying uncertain data, our work focuses on the process in which relational queries can produce partial results when failures occur during query processing. In other words, one can see a partial result-aware system that we have described here as populating the kinds of databases that these prior state-of-the-art works have described.

8. CONCLUSIONS AND FUTURE WORK

In this paper we deal with the problem of partial results, whereby query execution continues and results are produced even when some data may be unavailable. We identify partial result classes and present a classification framework equipped with four different models for analyzing partial results semantics at various granularities. The contributions of this paper can be utilized in large scale systems as either a mitigation mechanism for frequent failures or unexpected unavailability of data sources, as well as for exploratory, time-constrained query processing.

Substantial room for future work exists. For example, it would be interesting to explore partial results-aware query optimization, including the development of cost functions and plan selection algorithms that take into account user preferences regarding partial results semantics and the likelihood/impact of particular data access failures. Another direction to explore is a deeper investigation of the interaction between the schema, partitioning functions (including columnar storage and vectorized execution), queries, and specifics of incomplete inputs that may yield finer granularity reports on the quality of results. As discussed in the related work section, if our knowledge of the loosely coupled cloud databases is enhanced with constraints and dependencies, then we can provide even more precise guarantees. Finally, exploring alternatives for presenting partial result information to users is fertile ground for further research.

9. REFERENCES

[1] S. Acharya, P. B. Gibbons, V. Poosala, and S. Ramaswamy. Join Synopses for Approximate Query Answering. In *SIGMOD*, 1999.
[2] S. Agarwal, B. Mozafari, A. Panda, et al. BlinkDB: Queries with Bounded Errors and Bounded Response Times on Very Large Data. In *EuroSys*, 2013.
[3] P. Agrawal, O. Benjelloun, A. D. Sarma, et al. Trio: A system for data, uncertainty, and lineage. In *PVLDB*, 2006.
[4] T. Akidau, A. Balikov, K. Bekiroğlu, S. Chernyak, J. Haberman, et al. MillWheel: Fault-tolerant Stream Processing at Internet Scale. In *VLDB*, 2013.
[5] M. Arenas, L. Bertossi, and J. Chomicki. Consistent Query Answers in Inconsistent Databases. In *PODS*, 1999.
[6] S. Babu and J. Widom. Continuous Queries over Data Streams. *SIGMOD Record*, 2001.
[7] J. Barr, A. Narin, and J. Varia. Building Fault-Tolerant Applications on AWS. 2011. `http://bit.ly/1cD6k4w`.
[8] O. Benjelloun, A. Das Sarma, A. Halevy, M. Theobald, and J. Widom. Databases with Uncertainty and Lineage. *VLDB Journal*, 2008.
[9] P. Bonnet and A. Tomasic. Partial Answers for Unavailable Data Sources. *INRIA Technical Report*, 1997.
[10] Y. Cao, W. Fan, and W. Yu. Determining the Relative Accuracy of Attributes. In *SIGMOD*, 2013.
[11] K. Chakrabarti, M. Garofalakis, R. Rastogi, and K. Shim. Approximate Query Processing Using Wavelets. *VLDB Journal*, 2001.
[12] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, et al. TelegraphCQ: Continuous Dataflow Processing. In *SIGMOD*, 2003.
[13] A. Chapman and H. V. Jagadish. Why Not? In *SIGMOD*, 2009.
[14] R. Cheng, J. Chen, and X. Xie. Cleaning Uncertain Data with Quality Guarantees. In *VLDB*, 2008.
[15] R. Cheng, D. V. Kalashnikov, and S. Prabhakar. Evaluating Probabilistic Queries over Imprecise Data. In *SIGMOD*, 2003.
[16] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, et al. Spanner: Google's Globally-distributed Database. In *OSDI*, 2012.
[17] Y. Cui, J. Widom, and J. L. Wiener. Tracing the Lineage of View Data in a Warehousing Environment. *TODS*, 2000.
[18] N. Dalvi and D. Suciu. Management of Probabilistic Data: Foundations and Challenges. In *PODS*, 2007.
[19] D. Fabbri and K. LeFevre. Explanation-based Auditing. In *VLDB*, 2011.
[20] F. Geerts, G. Mecca, P. Papotti, and D. Santoro. The LLUNATIC Data-Cleaning Framework. In *PVLDB*, 2013.
[21] A. Gheerbrant, L. Libkin, and C. Sirangelo. When is Naive Evaluation Possible? In *PODS*, 2013.
[22] P. B. Gibbons and Y. Matias. New Sampling-Based Summary Statistics for Improving Approximate Query Answers. In *SIGMOD*, 1998.
[23] J. M. Hellerstein, P. J. Haas, and H. J. Wang. Online Aggregation. In *SIGMOD*, 1997.
[24] Y. Hu, S. Sundara, and J. Srinivasan. Supporting Time-constrained SQL Queries in Oracle. In *VLDB*, 2007.
[25] Y. E. Ioannidis and S. Christodoulakis. On the Propagation of Errors in the Size of Join Results. In *SIGMOD*, 1991.
[26] A. Meliou, W. Gatterbauer, et al. The Complexity of Causality and Responsibility for Query Answers and non-Answers. In *PVLDB*, 2010.
[27] S. Nirkhiwale, A. Dobra, and C. M. Jermaine. A Sampling Algebra for Aggregate Estimation. In *PVLDB*, 2013.
[28] N. Pansare, V. R. Borkar, C. Jermaine, and T. Condie. Online Aggregation for Large MapReduce Jobs. In *PVLDB*, 2011.
[29] H. Park, R. Ikeda, and J. Widom. RAMP: A System for Capturing and Tracing Provenance in MapReduce Workflows. In *PVLDB*, 2011.
[30] V. Raman and J. M. Hellerstein. Potter's Wheel: An Interactive Data Cleaning System. In *VLDB*, 2001.
[31] V. Raman and J. M. Hellerstein. Partial Results for Online Query Processing. In *SIGMOD*, 2002.
[32] C. Ré and D. Suciu. Approximate Lineage for Probabilistic Databases. In *VLDB*, 2008.
[33] J. Shanmugasundaram, K. Tufte, et al. Architecting a Network Query Engine for Producing Partial Results. In *WebDB*, 2000.
[34] J. Shute, R. Vingralek, B. Samwel, B. Handy, C. Whipkey, et al. F1: A Distributed SQL Database That Scales. In *VLDB*, 2013.
[35] H. Singh. Fault-tolerance in Windows Azure SQL Database. 2012. `http://bit.ly/1bUS3V5`.
[36] M. A. Soliman, I. F. Ilyas, and S. Ben-David. Supporting Ranking Queries on Uncertain and Incomplete Data. *VLDB Journal*, 2010.
[37] S. V. Vrbsky and J. W. S. Liu. APPROXIMATE – A Query Processor That Produces Monotonically Improving Approximate Answers. *TKDE*, 1993.
[38] T.-Y. Wang, C. Ré, and D. Suciu. Implementing NOT EXISTS Predicates over a Probabilstic Database. In *MUD*, 2008.
[39] J. Widom. Trio: A System for Integrated Management of Data, Accuracy, and Lineage. Technical report, Stanford InfoLab, 2004.
[40] J. Zhou, P. A. Larson, and R. Chaiken. Incorporating Partitioning and Parallel Plans into the SCOPE Optimizer. In *ICDE*, 2010.
[41] W. Zhou, S. Mapara, Y. Ren, Y. Li, A. Haeberlen, Z. Ives, B. T. Loo, and M. Sherr. Distributed time-aware provenance. In *PVLDB*, 2013.