Outerjoins in Uncertain Databases

Robert Ikeda and Jennifer Widom

Stanford University

Abstract. We consider the problem of incorporating outerjoins into uncertain databases. We motivate why outerjoins are useful, but tricky, in uncertain databases, arguing that standard possible-worlds semantics may be inappropriate for outerjoins. We explore a variety of alternative semantics through a running example, and we briefly discuss implementation considerations.

1 Introduction

Uncertain data appears in a number of applications including data integration, information extraction, and entity resolution. There has been a large amount of work recently on *uncertain database systems* for managing this data, e.g., [2–7]. Relational operations such as selection, projection, join, and aggregation have been defined and implemented for these systems. In traditional (certain) databases, there is a variation of the join operation, called *outerjoin*, that includes in its result "dangling" tuples, or those tuples that do not match any tuple of the other relation. We are unaware of any work that considers outerjoins in uncertain databases. Here, we motivate why outerjoins are useful, but tricky, in uncertain databases, and we explore various possible semantics for them.

Outerjoin is a widely used operation in traditional relational databases, because its result does not lose data when the join is "incomplete." With uncertain data, it seems even more likely that non-matching tuples will occur in a join, especially if the data is very noisy, or if the join condition is too strict. Thus, we expect outerjoins to be very useful in uncertain databases.

Unfortunately, it is not clear what the semantics of the outerjoin operator in an uncertain database should be. One obvious idea is to follow the *possible-worlds* semantics that is common for uncertain databases [3], but we will see that this semantics may be inappropriate for outerjoins. Through a running example we explore a variety of other possible semantics, illustrating some advantages and disadvantages of each.

1.1 Running Example

Consider an uncertain database containing information about which companies certain executives (CEOs and CFOs) work for. Suppose the information is extracted from unstructured sources, so it is uncertain; specifically, although executive names are consistent, they may be outdated, and company names may be

inconsistent. (Although this running example is contrived and simplistic for illustrative purposes, it is meant to be representative of information extraction and entity-resolution scenarios.) We have two tables, **CEO(ceo_name,company)** and **CFO(cfo_name,company)**. Sample contents of the two tables are as follows, where ‖ denotes *alternative values* for tuples [3], to be formalized in Section 2.

CEO (ceo_name, company)
Chambers, Cisco ‖ Chambers, Cisco Sys.
Schmidt, Google
Ellison, Oracle ‖ Ellison, Oracle Corp.
Ballmer, Microsoft ‖ Ballmer, Microsoft Corp.
Gates, Microsoft ‖ Gates, MSFT

CFO (cfo_name, company)
Calderoni, Cisco ‖ Calderoni, Cisco Sys.
Catz, Oracle
Jorgensen, Yahoo ‖ Jorgensen, Yahoo!
Liddell, Microsoft ‖ Liddell, Microsoft Corp.

Suppose we are interested in a natural join of the two tables on attribute **company**, but we don't want to lose CEO information for a company because we don't have CFO information, or vice-versa. Thus, we wish to compute a natural outerjoin of tables **CEO** and **CFO**.

First consider *possible-worlds* semantics. Here, the uncertain result must include every possible result obtained by choosing any combination of alternatives from the two input relations. In outerjoin, dangling (unmatched) tuples are "padded" with NULL values. Thus, using possible-worlds semantics, the natural outerjoin of **CEO** and **CFO** gives us:[1]

[1] As we will see in Section 2, we actually require *lineage* in addition to the data shown for correctness.

(company, ceo_name, cfo_name)
Cisco, Chambers, Calderoni ‖ Cisco Sys., Chambers, Calderoni
Oracle, Ellison, Catz
Microsoft, Ballmer, Liddell ‖ Microsoft Corp., Ballmer, Liddell
Microsoft, Gates, Liddell
Cisco, Chambers, NULL ‖ Cisco Sys., Chambers, NULL
Google, Schmidt, NULL
Oracle, Ellison, NULL ‖ Oracle Corp., Ellison, NULL
Microsoft, Ballmer, NULL ‖ Microsoft Corp., Ballmer, NULL
Microsoft, Gates, NULL ‖ MSFT, Gates, NULL
Cisco, NULL, Calderoni ‖ Cisco Sys., NULL, Calderoni
Oracle, NULL, Catz
Yahoo, NULL, Jorgensen ‖ Yahoo!, NULL, Jorgensen
Microsoft, NULL, Liddell ‖ Microsoft Corp., NULL, Liddell

Our feeling is that this result is cumbersome and nonintuitive. In general, outerjoin results using possible-worlds semantics are cluttered with NULLs, since every possible scenario of non-matching join tuples must be represented. A more intuitive result in this case is obtained using what we will call "inner-outerjoin semantics" (Section 5):

(company, ceo_name, cfo_name)
Cisco, Chambers, Calderoni ‖ Cisco Sys., Chambers, Calderoni
Oracle, Ellison, Catz
Microsoft, Ballmer, Liddell ‖ Microsoft Corp., Ballmer, Liddell
Microsoft, Gates, Liddell
Google, Schmidt, NULL
Yahoo, NULL, Jorgensen ‖ Yahoo!, NULL, Jorgensen

Although this result seems superior to the previous one, it may not always be ideal. We will explore two other alternatives to possible-worlds as well.

1.2 Remainder of Paper

In the remainder of this short paper, we first review the *ULDB* data model used by the Stanford *Trio* system [7] and as the basis for this work. We then formally apply Trio's possible-worlds semantics for outerjoin, motivating why other semantics for outerjoin may be more useful. Next, we define three possible alternative semantics, illustrating each one on our running example. We summarize and contrast the four semantics, and finally briefly discuss implementation considerations.

2 ULDB Data Model

We review Trio's *Uncertainty-Lineage Database* (*ULDB*) data model [7]. Each tuple in a ULDB relation consists of a set of mutually-exclusive *alternatives*.

Intuitively, the tuple takes the value of one of its alternatives. Consider again table **CEO** listing CEOs and the companies that they lead:

ID	**CEO (ceo_name, company)**
11	Chambers, Cisco ‖ Chambers, Cisco Sys.
12	Schmidt, Google
13	Ellison, Oracle ‖ Ellison, Oracle Corp.
14	Ballmer, Microsoft ‖ Ballmer, Microsoft Corp.
15	Gates, Microsoft ‖ Gates, MSFT

The ID column is added for identification purposes. We shall identify an alternative of a tuple by a pair (i, j), where i is the tuple ID, and j is the index of the alternative in tuple i. For instance, (Chambers, Cisco Sys.) is identified by (11,2). These identifiers are needed by lineage, as we shall see shortly.

ULDB relations follow the standard possible-worlds semantics: The possible-worlds of a relation (without lineage) are constructed by choosing exactly one alternative from each tuple. Each possible-world is a conventional relation. Note that the **CEO** table above has 16 possible-worlds.

In ULDBs, query result relations include *lineage* to the query's input relations. Lineage is critical for correctness, and is used in Trio for other features as well [7]. To show how lineage is generated by queries, let us add table **CFO**, now with IDs:

ID	**CFO (cfo_name, company)**
21	Calderoni, Cisco ‖ Calderoni, Cisco Sys.
22	Catz, Oracle
23	Jorgensen, Yahoo ‖ Jorgensen, Yahoo!
24	Liddell, Microsoft ‖ Liddell, Microsoft Corp.

Suppose we perform a standard natural (inner) join of tables **CEO** and **CFO** on attribute **company**. We obtain the following table:

ID	**(company, ceo_name, cfo_name)**
31	Cisco, Chambers, Calderoni ‖ Cisco Sys., Chambers, Calderoni
32	Oracle, Ellison, Catz
33	Microsoft, Ballmer, Liddell ‖ Microsoft Corp., Ballmer, Liddell
34	Microsoft, Gates, Liddell

$$\lambda(31,1) = (11,1) \wedge (21,1) \qquad \lambda(31,2) = (11,2) \wedge (21,2)$$
$$\lambda(32,1) = (13,1) \wedge (22,1)$$
$$\lambda(33,1) = (14,1) \wedge (24,1) \qquad \lambda(33,2) = (14,2) \wedge (24,2)$$
$$\lambda(34,1) = (15,1) \wedge (24,1)$$

The λ functions below the table capture *lineage*. For example, $\lambda(31,2) = (11,2) \wedge (21,2)$ says that the second alternative of result tuple 31 was derived from the

second alternative of **CEO** tuple 11 and the second alternative of **CFO** tuple 21. For complex queries, lineage formulas can be any boolean expression, and for derived relations the possible-worlds generated by the relation are dependent on lineage as well as on data. Full details are available in [3].

3 Semantics #1: Possible-Worlds

In Section 1.1 we already saw the ULDB relation produced by applying possible-worlds semantics to the natural outerjoin of **CEO** and **CFO**. Here is the table again, now with lineage:

ID	**(company, ceo_name, cfo_name)**
31	Cisco, Chambers, Calderoni ‖ Cisco Sys., Chambers, Calderoni
32	Oracle, Ellison, Catz
33	Microsoft, Ballmer, Liddell ‖ Microsoft Corp., Ballmer, Liddell
34	Microsoft, Gates, Liddell
35	Cisco, Chambers, NULL ‖ Cisco Sys., Chambers, NULL
36	Google, Schmidt, NULL
37	Oracle, Ellison, NULL ‖ Oracle Corp., Ellison, NULL
38	Microsoft, Ballmer, NULL ‖ Microsoft Corp., Ballmer, NULL
39	Microsoft, Gates, NULL ‖ MSFT, Gates, NULL
40	Cisco, NULL, Calderoni ‖ Cisco Sys., NULL, Calderoni
41	Oracle, NULL, Catz
42	Yahoo, NULL, Jorgensen ‖ Yahoo!, NULL, Jorgensen
43	Microsoft, NULL, Liddell ‖ Microsoft Corp., NULL, Liddell

$\lambda(31,1) = (11,1) \wedge (21,1)$ $\quad$ $\lambda(31,2) = (11,2) \wedge (21,2)$
$\lambda(32,1) = (13,1) \wedge (22,1)$
$\lambda(33,1) = (14,1) \wedge (24,1)$ $\quad$ $\lambda(33,2) = (14,2) \wedge (24,2)$
$\lambda(34,1) = (15,1) \wedge (24,1)$
$\lambda(35,1) = (11,1) \wedge \neg(21,1)$ $\quad$ $\lambda(35,2) = (11,2) \wedge \neg(21,2)$
$\lambda(36,1) = (12,1)$
$\lambda(37,1) = (13,1) \wedge \neg(22,1)$ $\quad$ $\lambda(37,2) = (13,2)$
$\lambda(38,1) = (14,1) \wedge \neg(24,1)$ $\quad$ $\lambda(33,2) = (14,2) \wedge \neg(24,2)$
$\lambda(39,1) = (15,1) \wedge \neg(24,1)$ $\quad$ $\lambda(39,2) = (15,2)$
$\lambda(40,1) = (21,1) \wedge \neg(11,1)$ $\quad$ $\lambda(40,2) = (21,2) \wedge \neg(11,2)$
$\lambda(41,1) = (22,1) \wedge \neg(13,1)$
$\lambda(42,1) = (23,1)$ $\quad$ $\lambda(42,2) = (23,2)$
$\lambda(43,1) = (24,1) \wedge \neg(14,1) \wedge \neg(15,1)$ $\quad$ $\lambda(43,2) = (24,2) \wedge \neg(14,2)$

Note that some of the lineage formulas include negation. For example, $\lambda(35,1) = (11,1) \wedge \neg(21,1)$ says that the first alternative of result tuple 35 is present in exactly those possible-worlds in which the first alternative of **CEO** tuple 11 is present and the first alternative of **CFO** tuple 21 is not present.

Compute $R = S_1 \overset{\circ}{\bowtie}_P^{PW} S_2$

Step 1: Compute innerjoin. For each pair of tuples s_1 in S_1 and s_2 in S_2, if at least one pair of alternatives satisfies P:

- Create a tuple t in R containing the "horizontal innerjoin" of s_1 and s_2: t contains an alternative a for every pair of alternatives $a_1 \in s_1$ and $a_2 \in s_2$ that satisfies P, with $\lambda(a) = a_1 \wedge a_2$.

Step 2: Add the left relation right-padded with NULLs. For each tuple s_1 in S_1:

- Create a tuple t in R containing an alternative $a = a_1 \cdot \text{NULL}$ for each alternative a_1 in s_1, with $\lambda(a) = a_1 \wedge \neg a_{2_1} \wedge \neg a_{2_2} \wedge ... \wedge \neg a_{2_m}$, where the a_{2_i} are all alternatives in S_2 such that a_1 and a_{2_i} satisfy P.

Step 3: Add the right relation left-padded with NULLs. For each tuple s_2 in S_2:

- Create a tuple t in R containing an alternative $a = \text{NULL} \cdot a_2$ for each alternative a_2 in s_2, with $\lambda(a) = a_2 \wedge \neg a_{1_1} \wedge \neg a_{1_2} \wedge ... \wedge \neg a_{1_n}$, where the a_{1_i} are all alternatives in S_1 such that a_{1_i} and a_2 satisfy P.

Fig. 1. Algorithm for possible-worlds outerjoin.

Figure 1 specifies an algorithm that computes the possible-worlds outerjoin, which we denote by operator $\overset{\circ}{\bowtie}^{PW}$. (For simplicity and efficiency, like in Trio [7], our algorithm produces the correct result but does not automatically remove *extraneous data*, i.e., data that cannot appear in any possible-world.) In all of our algorithms, we use "a·NULL" to create tuples from the left relation of the join right-padded with NULLs, and similarly "NULL·a" for left-padded tuples from the right relation.

Note that although the result table in our example may appear to have $2^9 = 512$ possible-worlds, lineage constrains it to have only 128, as we would expect from the 16 possible-worlds of **CEO** combined with the 8 possible-worlds of **CFO**. Again, the reader is referred to [3] for details.

The result above is correct with respect to possible-worlds semantics, and therefore can be integrated seamlessly with other query operators. However, as discussed earlier, our feeling is that this result is unnecessarily cluttered with NULLs and therefore not very intuitive or useful. We will now explore more intuitive and compact results that still capture the "no lost data" spirit of outerjoin.

4 Semantics #2: Outer-Outerjoin

Our first alternative semantics is referred to as *outer-outerjoin* and denoted by operator $\overset{\circ}{\bowtie}^{O}$. Ideally, instead of including in the result a tuple t padded with NULLs whenever t may be dangling in *some* possible-world, we would like to only include t padded with NULLs whenever t is dangling in *every* possible-

world. Unfortunately, this property is very expensive to guarantee in general (essentially requiring an enumeration of possible-worlds), but it is approximated by the outer-outerjoin and does hold in many cases.

Here is the result of this semantics on our running example:

ID	**(company, ceo_name, cfo_name)**
31	Cisco, Chambers, Calderoni ‖ Cisco Sys., Chambers, Calderoni
32	Oracle, Ellison, Catz ‖ Oracle Corp., Ellison, NULL
33	Microsoft, Ballmer, Liddell ‖ Microsoft Corp., Ballmer, Liddell
34	Microsoft, Gates, Liddell ‖ MSFT, Gates, NULL ‖ Microsoft Corp., NULL, Liddell
35	Google, Schmidt, NULL
36	Yahoo, NULL, Jorgensen ‖ Yahoo!, NULL, Jorgensen

$\lambda(31,1) = (11,1) \wedge (21,1)$ $\quad$ $\lambda(31,2) = (11,2) \wedge (21,2)$
$\lambda(32,1) = (13,1) \wedge (22,1)$ $\quad$ $\lambda(32,2) = (13,2) \wedge (22,$ NULL$)$
$\lambda(33,1) = (14,1) \wedge (24,1)$ $\quad$ $\lambda(33,2) = (14,2) \wedge (24,2)$
$\lambda(34,1) = (15,1) \wedge (24,1)$ $\quad$ $\lambda(34,2) = (15,2) \wedge (24,$ NULL$)$
$\lambda(34,3) = (24,2) \wedge (15,$ NULL$)$
$\lambda(35,1) = (12,1)$
$\lambda(36,1) = (23,1)$ $\quad$ $\lambda(36,2) = (23,2)$

Refer to Figure 2 for an algorithm that computes the outer-outerjoin. Notice that NULL-padded alternatives in tuples that do join are assigned a special type of NULL lineage indicating a non-match in the horizontal "outerjoin" (Steps 1(ii) and 1(iii) in the algorithm)—see $\lambda(32,2)$ for example.

It should be obvious that, on our running example at least, this operator produces a result that is considerably more intuitive than the possible-worlds result, while still retaining the property that no tuples from either input relation are lost.

5 Semantics #3: Inner-Outerjoin

Although the outer-outerjoin operator is a significant improvement over the possible-worlds outerjoin, it still contains some NULL-padded results that may be considered unnecessary. Consider for example the second alternative of result tuple 32: (Oracle Corp., Ellison, NULL), derived from **CEO** tuple 13 and **CFO** tuple 22. Within the alternatives of these joining tuples, we have a match on "Oracle," so we might prefer to assume "Oracle" is the correct value, and omit the (Oracle Corp., Ellison, NULL) alternative. Similarly, consider the second and third alternatives of result tuple 34: (MSFT, Gates, NULL) and (Microsoft Corp., NULL, Liddell). This result tuple is derived from **CEO** tuple 15 and **CFO** tuple 24. Here, we have an exact match on "Microsoft," so again we may prefer to omit the alternatives without matches. The next semantics, which we refer to as *inner-outerjoin* and denote by operator $\overset{\circ}{\bowtie}^I$, is based on this intuition.

Compute $R = S_1 \overset{\circ}{\bowtie}{}^{O}_{P} S_2$

Step 1: For each pair of tuples s_1 in S_1 and s_2 in S_2, if at least one pair of alternatives satisfies P:

- Create a tuple t in R containing the "horizontal outerjoin" of s_1 and s_2:
  - (i) Add an alternative a to t for every pair of alternatives $a_1 \in s_1$ and $a_2 \in s_2$ that satisfies P, with $\lambda(a) = a_1 \wedge a_2$.
  - (ii) For every $a_1 \in s_1$ not used in step (i), add $a = a_1 \cdot$NULL to t and set $\lambda(a) = a_1 \wedge (s_2.\text{ID}, \text{NULL})$.
  - (iii) For every $a_2 \in s_2$ not used in step (i), add $a = \text{NULL} \cdot a_2$ to t and set $\lambda(a) = a_2 \wedge (s_1.\text{ID}, \text{NULL})$.

Step 2: Add the left relation's dangling tuples right-padded with NULLs. For each tuple s_1 in S_1 not used in Step 1:

- Create a tuple t in R containing an alternative $a = a_1 \cdot$NULL for each alternative a_1 in s_1, with $\lambda(a) = a_1$.

Step 3: Add the right relation's dangling tuples left-padded with NULLs. For each tuple s_2 in S_2 not used in Step 1:

- Create a tuple t in R containing an alternative $a = \text{NULL} \cdot a_2$ for each alternative a_2 in s_2, with $\lambda(a) = a_2$.

Fig. 2. Algorithm for outer-outerjoin.

Figure 3 contains an algorithm for computing the inner-outerjoin. As can be seen, it combines Step 1 from the possible-worlds outerjoin with Steps 2 and 3 from the outer-outerjoin. We saw the result of this semantics on our running example in Section 1.1; here it is again, now with lineage:

ID	**(company, ceo_name, cfo_name)**
31	Cisco, Chambers, Calderoni ‖ Cisco Sys., Chambers, Calderoni
32	Oracle, Ellison, Catz
33	Microsoft, Ballmer, Liddell ‖ Microsoft Corp., Ballmer, Liddell
34	Microsoft, Gates, Liddell
35	Google, Schmidt, NULL
36	Yahoo, NULL, Jorgensen ‖ Yahoo!, NULL, Jorgensen

$\lambda(31,1) = (11,1) \wedge (21,1)$ $\quad$ $\lambda(31,2) = (11,2) \wedge (21,2)$
$\lambda(32,1) = (13,1) \wedge (22,1)$
$\lambda(33,1) = (14,1) \wedge (24,1)$ $\quad$ $\lambda(33,2) = (14,2) \wedge (24,2)$
$\lambda(34,1) = (15,1) \wedge (24,1)$
$\lambda(35,1) = (12,1)$
$\lambda(36,1) = (23,1)$ $\quad$ $\lambda(36,2) = (23,2)$

Compute $R = S_1 \overset{\circ}{\bowtie}^I_P S_2$

Step 1: Compute innerjoin. For each pair of tuples s_1 in S_1 and s_2 in S_2, if at least one pair of alternatives satisfies P:

- Create a tuple t in R containing the "horizontal innerjoin" of s_1 and s_2: t contains an alternative a for every pair of alternatives $a_1 \in s_1$ and $a_2 \in s_2$ that satisfies P, with $\lambda(a) = a_1 \wedge a_2$.

Step 2: Add the left relation's dangling tuples right-padded with NULLs. For each tuple s_1 in S_1 not used in Step 1:

- Create a tuple t in R containing an alternative $a = a_1 \cdot \text{NULL}$ for each alternative a_1 in s_1, with $\lambda(a) = a_1$.

Step 3: Add the right relation's dangling tuples left-padded with NULLs. For each tuple s_2 in S_2 not used in Step 1:

- Create a tuple t in R containing an alternative $a = \text{NULL} \cdot a_2$ for each alternative a_2 in s_2, with $\lambda(a) = a_2$.

Fig. 3. Algorithm for inner-outerjoin.

There is a tradeoff between inner-outerjoin and outer-outerjoin. Inner-outerjoin semantics effectively says that if a tuple t has at least one alternative satisfying the join condition, then alternatives in t that never satisfy the join condition can be dropped. While the result is more compact, it is our first semantics in which some values from the input relation do not appear in the result (although all tuples still appear in some form). In our running example, values "Oracle Corp." and "MSFT" from the input relations do not appear at all in the result.

6 Semantics #4: Cross-Outerjoin

Our last semantics is referred to as *cross-outerjoin* and denoted by operator $\overset{\circ}{\bowtie}^X$. Now, instead of using a "horizontal outerjoin" (Semantics #2) or "horizontal innerjoin" (Semantics #3) for matching tuples, this semantics uses a "horizontal cross-product."

Refer to Figure 4 for an algorithm that computes the cross-outerjoin. Note that since alternatives in the horizontal cross-product do not always agree on the join attribute **company**, we can no longer compute a natural join for our running example—we must retain the join attributes from the two input relations separately. Here is the result of this semantics on our running example:

Compute $R = S_1 \overset{\circ}{\bowtie}{}^{X}_{P} S_2$

Step 1: For each pair of tuples s_1 in S_1 and s_2 in S_2, if at least one pair of alternatives satisfies P:

- Create a tuple t in R with the "horizontal cross-product" of s_1 and s_2: t contains an alternative a for every pair of alternatives $a_1 \in s_1$ and $a_2 \in s_2$, with $\lambda(a) = a_1 \wedge a_2$.

Step 2: Add the left relation's dangling tuples right-padded with NULLs. For each tuple s_1 in S_1 not used in Step 1:

- Create a tuple t in R containing an alternative $a = a_1 \cdot \text{NULL}$ for each alternative a_1 in s_1, with $\lambda(a) = a_1$.

Step 3: Add the right relation's dangling tuples left-padded with NULLs. For each tuple s_2 in S_2 not used in Step 1:

- Create a tuple t in R containing an alternative $a = \text{NULL} \cdot a_2$ for each alternative a_2 in s_2, with $\lambda(a) = a_2$.

Fig. 4. Algorithm for cross-outerjoin.

ID	**(ceo_company, ceo_name, cfo_company, cfo_name)**
31	Cisco, Chambers, Cisco, Calderoni ‖ Cisco, Chambers, Cisco Sys., Calderoni ‖ Cisco Sys., Chambers, Cisco, Calderoni ‖ Cisco Sys., Chambers, Cisco Sys., Calderoni
32	Oracle, Ellison, Oracle, Catz ‖ Oracle Corp., Ellison, Oracle, Catz
33	Microsoft, Ballmer, Microsoft, Liddell ‖ Microsoft, Ballmer, Microsoft Corp., Liddell ‖ Microsoft Corp., Ballmer, Microsoft, Liddell ‖ Microsoft Corp., Ballmer, Microsoft Corp., Liddell
34	Microsoft, Gates, Microsoft, Liddell ‖ Microsoft, Gates, Microsoft Corp., Liddell ‖ MSFT, Gates, Microsoft, Liddell ‖ MSFT, Gates, Microsoft Corp., Liddell
35	Google, Schmidt, NULL, NULL
36	NULL, NULL, Yahoo, Jorgensen ‖ NULL, NULL, Yahoo!, Jorgensen

$$\begin{array}{ll}
\lambda(31,1) = (11,1) \wedge (21,1) & \lambda(31,2) = (11,1) \wedge (21,2) \\
\lambda(31,3) = (11,2) \wedge (21,1) & \lambda(31,4) = (11,2) \wedge (21,2) \\
\lambda(32,1) = (13,1) \wedge (22,1) & \lambda(32,2) = (13,2) \wedge (22,1) \\
\lambda(33,1) = (14,1) \wedge (24,1) & \lambda(33,2) = (14,1) \wedge (24,2) \\
\lambda(33,3) = (14,2) \wedge (24,1) & \lambda(33,4) = (14,2) \wedge (24,2) \\
\lambda(34,1) = (15,1) \wedge (24,1) & \lambda(34,2) = (15,1) \wedge (24,2) \\
\lambda(34,3) = (15,2) \wedge (24,1) & \lambda(34,4) = (15,2) \wedge (24,2) \\
\lambda(35,1) = (12,1) & \\
\lambda(36,1) = (23,1) & \lambda(36,2) = (23,2)
\end{array}$$

This result may appear cumbersome and not particularly useful. However, if we follow the cross-outerjoin with additional processing based on Trio's *horizontal subqueries* (intuitively, subqueries that treat the alternatives of a tuple as a relation [7]), we get a more intuitive result. Specifically, for each tuple t whose lineage includes both of the input relations (i.e., tuples produced by Step 1 of the algorithm), do the following: Treat t's alternatives as a relation, project once dropping the join attributes from the right input relation and once dropping the join attributes from the left input relation, and union the results without retaining duplicates. For the tuples whose lineage includes just one input relation (i.e., tuples produced by Steps 2 and 3 of the algorithm), simply project away the (NULL) join attributes from the other relation. Although it may sound complex, this processing is not difficult to compute, and together with the cross-outerjoin it yields the following result:

ID	**(company, ceo_name, cfo_name)**
31	Cisco, Chambers, Calderoni ‖ Cisco Sys., Chambers, Calderoni
32	Oracle, Ellison, Catz ‖ Oracle Corp., Ellison, Catz
33	Microsoft, Ballmer, Liddell ‖ Microsoft Corp., Ballmer, Liddell
34	Microsoft, Gates, Liddell ‖ Microsoft Corp., Gates, Liddell ‖ MSFT, Gates, Liddell
35	Google, Schmidt, NULL
36	Yahoo, NULL, Jorgensen ‖ Yahoo!, NULL, Jorgensen

$\lambda(31,1) = ((11,1) \wedge (21,1)) \vee ((11,1) \wedge (21,2)) \vee ((11,2) \wedge (21,1))$
$\lambda(31,2) = ((11,1) \wedge (21,2)) \vee ((11,2) \wedge (21,1)) \vee ((11,2) \wedge (21,2))$
$\lambda(32,1) = ((13,1) \wedge (22,1)) \vee ((13,2) \wedge (22,1))$
$\lambda(32,2) = (13,2) \wedge (22,1)$
$\lambda(33,1) = ((14,1) \wedge (24,1)) \vee ((14,1) \wedge (24,2)) \vee ((14,2) \wedge (24,1))$
$\lambda(33,2) = ((14,1) \wedge (24,2)) \vee ((14,2) \wedge (24,1)) \vee ((14,2) \wedge (24,2))$
$\lambda(34,1) = ((15,1) \wedge (24,1)) \vee ((15,1) \wedge (24,2)) \vee ((15,2) \wedge (24,1))$
$\lambda(34,2) = ((15,1) \wedge (24,2)) \vee ((15,2) \wedge (24,2))$
$\lambda(34,3) = ((15,2) \wedge (24,1)) \vee ((15,2) \wedge (24,2))$
$\lambda(35,1) = (12,1)$
$\lambda(36,1) = (23,1)$
$\lambda(36,2) = (23,2)$

Notice that we now have *disjunctive lineage*, which is generated by Trio when duplicate-elimination occurs [7].

This result may be our most intuitive of all, combining the best aspects of outer-outerjoin and inner-outerjoin. Specifically, it minimizes NULLs in the result, while not losing any values from the input data. In fact, we have recently been working on a new variant of Trio called *Trio-ER* [1], a workbench for entity-resolution. In Trio-ER, we have implemented operators whose behavior is very similar to the cross-outerjoin plus post-processing shown above.

7 Conclusions and Implementation

To summarize, we contrast the four semantics we have considered:

- The primary advantage of possible-worlds outerjoin is its formal foundations, and its adherence to the underlying semantics of Trio and other uncertain database systems. Its disadvantage, obviously, is the unwieldy nature of the results.
- Outer-outerjoin and inner-outerjoin vastly reduce the result size compared with possible-worlds, at the loss of a formal underlying semantics. The primary difference between the two is that outer-outerjoin guarantees to preserve all *values* (not just all *tuples*) present in the input relations, while inner-outerjoin does not. However, inner-outerjoin is more compact, and is appropriate in an entity-resolution scenario if values that agree are always thought to override those that don't.
- The final semantics we presented, cross-outerjoin, is not very intuitive as a stand-alone operator. However, when additional operations are applied as shown in Section 6, it may in fact be the most useful semantics in practice.

Let us briefly discuss implementation of the different semantics. As with outerjoin in conventional database systems, all of our semantics require a two-pass approach where dangling tuples are added after the innerjoin (or variant) is complete. Outer-outerjoin, inner-outerjoin, and cross-outerjoin differ only in how they combine matching tuples. Since we don't expect tuples to have large numbers of alternatives, these differences would probably amount to negligible in-memory processing. Step 1 in the possible-worlds semantics is similar, but Steps 2 and 3 require considerable additional processing or bookkeeping to generate correct lineage.

Considering the Trio system specifically, its implementation is layered over a conventional relational DBMS. Therefore, Trio queries are not executed by a special-purpose processor. Instead, they are translated to conventional SQL queries and operations over encoded ULDB relations [7]. All four of our semantics can be translated in the Trio system, with some requiring more complex SQL constructs than others. As an example, the following single query computes all three steps of the inner-outerjoin on our running example. (For Trio data encoding details such as the `xid` attribute, see [7]. Also, lineage must be computed separately, as in all Trio queries [7].) Note that the underlying SQL query processor would still typically require two passes for the SQL outerjoin included in this query, as discussed above.

```
Select E.company, E.ceo_name, F.cfo_name
From CEO_trans as E Full Outer Join CFO_trans as F
     On Exists (Select * From CEO_trans as E2, CFO_trans as F2
                Where E2.xid = E.xid and F2.xid = F.xid
                And E2.company = F2.company)
Where E.company = F.company
And E.company Is Not Null And F.company Is Not Null
```

The translation for cross-outerjoin simply omits the `Where` clause in the query above, while outer-outerjoin and possible-worlds outerjoin are more complex.

Acknowledgements. We would like to thank Hyunjung Park for several helpful discussions.

References

1. P. Agrawal, R. Ikeda, H. Park, and J. Widom. Trio-ER: The Trio system as a workbench for entity-resolution. Technical report, Stanford University InfoLab, March 2009.
2. L. Antova, T. Jansen, C. Koch, and D. Olteanu. Fast and simple relational processing of uncertain data. In *Proc. of Intl. Conference on Data Engineering (ICDE)*, Cancun, Mexico, Apr. 2008.
3. O. Benjelloun, A. Das Sarma, A. Y. Halevy, M. Theobald, and J. Widom. Databases with uncertainty and lineage. *The VLDB Journal*, 17(2):243–264, 2008.
4. N. Dalvi and D. Suciu. Efficient query evaluation on probabilistic databases. *The VLDB Journal*, 16(4):523–544, 2007.
5. P. Sen and A. Deshpande. Representing and querying correlated tuples in probabilistic databases. In *Proc. of Intl. Conference on Data Engineering (ICDE)*, Istanbul, Turkey, Apr. 2007.
6. S. Singh, C. Mayfield, S. Mittal, S. Prabhakar, S. Hambrusch, and R. Shah. Orion 2.0: native support for uncertain data. In *Proc. of ACM SIGMOD Intl. Conference on Management of Data*, Vancouver, Canada, June 2008.
7. J. Widom. Trio: A system for data, uncertainty, and lineage. In C. C. Aggarwal, editor, *Managing and Mining Uncertain Data*. Springer, 2008.