Demonstration of the FDB Query Engine for Factorised Databases

Nurzhan Bakibayev, Dan Olteanu, and Jakub Závodný
Department of Computer Science, University of Oxford, OX1 3QD, UK
{nurzhan.bakibayev, dan.olteanu, jakub.zavodny}@cs.ox.ac.uk

ABSTRACT

FDB is an in-memory query engine for factorised databases, which are relational databases that use compact factorised representations at the physical layer to reduce data redundancy and boost query performance.

We demonstrate FDB using real data sets from IMDB, DBLP, and the NELL repository of facts learned from Web pages. The users can inspect factorisations as well as plans used by FDB to compute factorised results of select-project-join queries on factorised databases.

1. FACTORISED DATABASES

The thesis underlying factorised databases is that relational databases can admit compact representations by algebraic factorisation using distributivity of product over union. This is similar in spirit to the relationship between logic functions in disjunctive normal form and their equivalent nested forms obtained by algebraic factorisation. In earlier work [7] we give a complete characterisation of the compactness of factorised results for select-project-join queries on relational databases and show that the gap between the sizes of query results and of their factorised representations can be exponential. In particular, there are arbitrarily large queries for which the query results have sizes exponential in the query size yet their factorised representations only have sizes bounded by the input database size. A similar exponential gap holds between the times needed to compute from the input relational database the query results and their factorised representations. Furthermore, the succinctness and performance gaps widen when we consider factorised databases as input. Experiments with our in-memory engine FDB for select-project-join queries on factorised databases show that FDB can be up to six orders of magnitude faster than relational engines such as PostgreSQL, SQLite, and a home-bred in-memory relational engine, for a wide range of queries on data sets with many-to-many relationships [3].

Factorised databases have applications beyond relational query evaluation. Factorised provenance polynomials are used as compact encoding of provenance [6] and for efficient query evaluation in probabilistic databases [8]. Factorised representations are a natural fit whenever we deal with a large space of possibilities and can be used to represent, e.g., AND/OR trees used in design specification [5] and world-set decompositions used for incomplete information [1]. They can also be used to compactly represent the space of feasible solutions to configuration problems in constraint satisfaction, where we need to connect a set of components so as to meet an objective while respecting given constraints [2].

The focus of this demonstration is our query engine FDB. The audience will experiment with FDB on several data sets including the NELL knowledge base learned from a large corpus of Web pages [4], will explore visually FDB evaluation plans as well as factorised intermediate and final query results, and will compare the time and space requirements of FDB to those of PostgreSQL and SQLite. FDB will be introduced to the audience by examples such as those in Section 2. The emphasis of the demonstration will be on FDB's evaluation and optimisation techniques, in particular on (1) the evaluation plans with novel operators to restructure factorisations, and on (2) the interplay of the standard optimisation objective of minimising the overall computation time and of the new objective of computing small factorised representations of query results.

2. FDB BY EXAMPLES

We introduce factorised databases and FDB by examples using the NELL data set. The demonstration will feature the examples from this section and further scenarios using NELL as well as IMDB and DBLP data sets. We will also show how FDB can manage solutions to crossword puzzles that are gradually refined as clues are supplied.

Sports Scenario. The NELL data set contains a myriad of small-size unary relations about players, teams, sports leagues and stadiums, and binary relations such as PlaysFor (players play for teams), CompetesIn (teams play in leagues), LeagueStadium (leagues played on stadiums), BasedIn and IsIn (teams and stadiums located in cities). We exemplify with queries that join these relations to compute (Q_1) players, teams and leagues that one can see playing on stadiums, (Q_2) teams and stadiums co-located in the same city, and (Q_3) players playing on stadiums in their home cities. The NELL binary relations are in general many-to-many. This makes the factorised results of our queries very compact when compared to flat relational representations. In the sequel, we discuss challenges in computing factorised results of our queries on input relational or factorised databases.

PlaysFor

player	team
Messi	Barcelona
Villa	Barcelona
Cech	Chelsea
Torres	Chelsea
van Persie	Arsenal

CompetesIn

team	league
Barcelona	Primera
Barcelona	Champions
Chelsea	Premier
Chelsea	Champions
Arsenal	Premier

LeagueStadium

league	stadium
Primera	CampNou
Champions	CampNou
Champions	Wembley
Premier	Stamford
Premier	Wembley

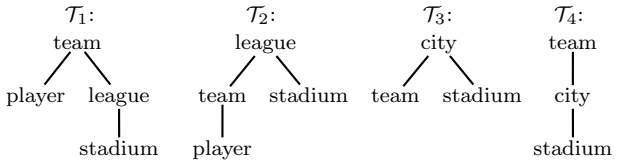
BasedIn

team	city
Barcelona	Barcelona
Chelsea	London
Arsenal	London

IsIn

stadium	city
CampNou	Barcelona
Wembley	London
Stamford	London

Figure 1: A sample from the NELL database. A constantly evolving database is available at rtw.ml.cmu.edu.



$Q_1 = \text{PlaysFor} \bowtie_{\text{team}} \text{CompetesIn} \bowtie_{\text{league}} \text{LeagueStadium}$

$Q_2 = \text{BasedIn} \bowtie_{\text{city}} \text{IsIn}$

Figure 2: Factorisation trees $\mathcal{T}_1$ and $\mathcal{T}_2$ for query Q_1 and $\mathcal{T}_3$ and $\mathcal{T}_4$ for query Q_2 used in the examples.

Factorised Query Results. The result to Q_1 on the sample database from Figure 1 is partially given below:

$Q_1 = \text{PlaysFor} \bowtie_{\text{team}} \text{CompetesIn} \bowtie_{\text{league}} \text{LeagueStadium}$

player	team	league	stadium
Messi	Barcelona	Primera	CampNou
Messi	Barcelona	Champions	CampNou
Messi	Barcelona	Champions	Wembley
Villa	Barcelona	Primera	CampNou
Villa	Barcelona	Champions	CampNou
		...	

In the result of Q_1, each player of each team is paired with each league the team competes in and each stadium of that league. We can obtain a more compact representation by factoring out these repeating occurrences:

$$\begin{aligned}
&\langle\text{Barcelona}\rangle \times (\langle\text{Messi}\rangle \cup \langle\text{Villa}\rangle) \times \\
&\quad \times (\langle\text{Primera}\rangle \times \langle\text{CampNou}\rangle \cup \\
&\quad\quad \langle\text{Champions}\rangle \times (\langle\text{CampNou}\rangle \cup \langle\text{Wembley}\rangle)) \cup \\
&\langle\text{Chelsea}\rangle \times (\langle\text{Cech}\rangle \cup \langle\text{Torres}\rangle) \times \\
&\quad \times (\langle\text{Premier}\rangle \times (\langle\text{Stamford}\rangle \cup \langle\text{Wembley}\rangle)) \cup \\
&\quad\quad \langle\text{Champions}\rangle \times (\langle\text{CampNou}\rangle \cup \langle\text{Wembley}\rangle)) \cup \\
&\langle\text{Arsenal}\rangle \times \langle\text{van Persie}\rangle \times \\
&\quad \times (\langle\text{Premier}\rangle \times (\langle\text{Stamford}\rangle \cup \langle\text{Wembley}\rangle)).
\end{aligned}$$

This factorisation has a regular nesting structure. For each team, we represent the union of its players independently of the union of its leagues with stadiums, and multiply them together. This nesting structure is captured by the tree $\mathcal{T}_1$ in Figure 2, which is called a *factorisation tree*, or f-tree for short. The f-tree $\mathcal{T}_2$ offers an alternative factorisation structure, where we first group by league and only then by teams with players and independently by stadiums:

$$\begin{aligned}
&\langle\text{Primera}\rangle \times \langle\text{Barcelona}\rangle \times (\langle\text{Messi}\rangle \cup \langle\text{Villa}\rangle) \times \langle\text{CampNou}\rangle \cup \\
&\langle\text{Champions}\rangle \times (\langle\text{Barcelona}\rangle \times (\langle\text{Messi}\rangle \cup \langle\text{Villa}\rangle) \cup \\
&\quad \langle\text{Chelsea}\rangle \times (\langle\text{Cech}\rangle \cup \langle\text{Torres}\rangle)) \times \\
&\quad \times (\langle\text{CampNou}\rangle \cup \langle\text{Wembley}\rangle) \cup \\
&\langle\text{Premier}\rangle \times (\langle\text{Chelsea}\rangle \times (\langle\text{Cech}\rangle \cup \langle\text{Torres}\rangle) \cup \\
&\quad \langle\text{Arsenal}\rangle \times \langle\text{van Persie}\rangle)) \times \\
&\quad \times (\langle\text{Stamford}\rangle \cup \langle\text{Wembley}\rangle).
\end{aligned}$$

Which of these factorisations is preferable depends on multiple factors such as the cost of subsequent workload of queries on them and most notably their size. For the latter, it is not realistic to compute all possible factorisations only to choose the smallest one. The choice of factorisation structure (f-tree) is thus guided by measures computed from the query and cardinality estimates available in the system catalogue. In earlier work [7] we introduce a parameter $s(\mathcal{T})$ for f-trees $\mathcal{T}$ which characterises the asymptotic behaviour of factorisation size with increasing database size. For example, in our case $s(\mathcal{T}_1) = s(\mathcal{T}_2) = 2$, which means that both factorisations are at most quadratic in the database size for *any* database. An alternative approach is to estimate the factorisation size using join selectivities and cardinality of input relations [3]. FDB implements both measures.

The result of Q_2 can be factorised over the f-tree $\mathcal{T}_3$ in Figure 2 as follows:

$$\begin{aligned}
&\langle\text{Barcelona}\rangle \times \langle\text{Barcelona}\rangle \times \langle\text{CampNou}\rangle \cup \\
&\langle\text{London}\rangle \times (\langle\text{Chelsea}\rangle \cup \langle\text{Arsenal}\rangle) \times (\langle\text{Wembley}\rangle \cup \langle\text{Stamford}\rangle).
\end{aligned}$$

This factorisation lists for each city the teams and the stadiums located in that city: there are one team and one stadium in Barcelona and two teams and two stadiums in London. A less compact factorisation is over the f-tree $\mathcal{T}_4$ in Figure 2: for each team, we list the cities of that team, and for each team and city, we list stadiums in that city:

$$\begin{aligned}
&\langle\text{Barcelona}\rangle \times \langle\text{Barcelona}\rangle \times \langle\text{CampNou}\rangle \cup \\
&\langle\text{Chelsea}\rangle \times \langle\text{London}\rangle \times (\langle\text{Wembley}\rangle \cup \langle\text{Stamford}\rangle) \cup \\
&\langle\text{Arsenal}\rangle \times \langle\text{London}\rangle \times (\langle\text{Wembley}\rangle \cup \langle\text{Stamford}\rangle).
\end{aligned}$$

This second factorisation does not exploit the join dependency in the query result, i.e., that for each city, the possible combinations of teams and stadiums in that city can be factorised compactly as the product of all these teams and of all these stadiums.

Queries on Factorised Databases. FDB can also evaluate queries on factorised databases using so-called factorisation plans [3]. Besides selection, projection, and join operators, such plans may also use operators for restructuring factorisations. In contrast to the relational case, the FDB's join operators are restricted; given an input factorised database over an f-tree $\mathcal{T}$, they can only be applied to nodes that are either siblings or in an ancestor-descendant relationship in $\mathcal{T}$. The reason for this restriction is efficiency; both join cases can be implemented efficiently using merge-sort join and do not require restructuring. If none of the two conditions hold for $\mathcal{T}$, then $\mathcal{T}$ has to be restructured so that one of the join cases becomes applicable. The restructuring operators are thus essential in enabling joins. We next illustrate joins and restructuring and then projections.

Let us continue our example and suppose that we would like to limit the result of Q_1 to those players based in the same city as the stadium they play on. This can be done by joining the result of Q_1, which lists players, teams and leagues playing on given stadiums, and the result of Q_2, which lists co-located teams and stadiums. This query is $Q_3 = Q_1 \bowtie_{\text{team,stadium}} Q_2$. We next consider two plans for

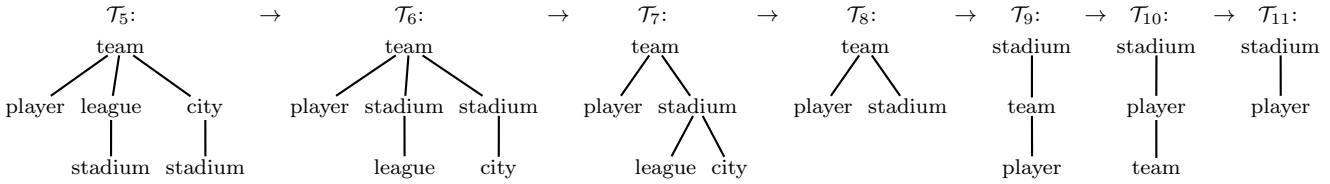


Figure 3: Factorisation plan for Q_3. The f-tree $\mathcal{T}_5$ is obtained by joining $\mathcal{T}_1$ with $\mathcal{T}_4$ on team; then $\mathcal{T}_6$ by swapping the nodes stadium with their parents; $\mathcal{T}_7$ by merging the nodes stadium; $\mathcal{T}_8$ by projection; $\mathcal{T}_9$ by swapping stadium and team; $\mathcal{T}_{10}$ by swapping team and player; $\mathcal{T}_{11}$ by final projection.

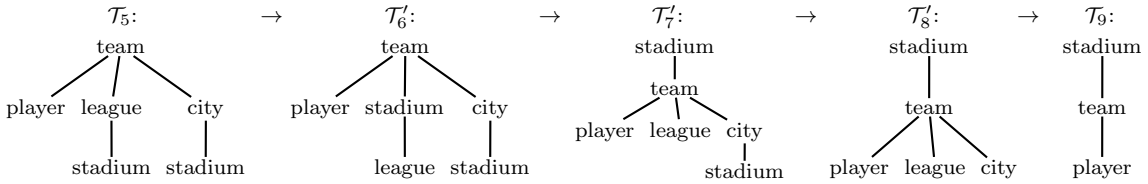


Figure 4: Alternative factorisation plan for Q_3. The f-tree $\mathcal{T}_6'$ is obtained from $\mathcal{T}_5$ by swapping the nodes stadium and league; then $\mathcal{T}_7'$ by swapping stadium with team; $\mathcal{T}_8'$ by absorbing the lower node stadium into the upper one; and $\mathcal{T}_9$ by projection. The plan then continues with $\mathcal{T}_{10}$ and $\mathcal{T}_{11}$ as in Figure 3.

Q_3. They are not (asymptotically) optimal and only used here to highlight FDB operations on factorisations.

If we assume that the f-tree of Q_1's result is $\mathcal{T}_1$ and the f-tree of Q_2's result is $\mathcal{T}_3$, then to compute Q_3, we would first need to restructure the input factorisations such that the joins can be performed. For the join on team, one possibility is to restructure the result of Q_2 such that it follows the structure of the f-tree $\mathcal{T}_4$. This can be obtained by *swapping* the parent node city with the child node team in $\mathcal{T}_3$ and performing the corresponding operation on the factorisation over $\mathcal{T}_3$. FDB has a swap operator to this effect. The new factorisation is now over $\mathcal{T}_4$ and is given above.

The factorisation over $\mathcal{T}_4$ can now be joined with the factorisation of Q_1's result over $\mathcal{T}_1$, as they are both grouped by teams. We simply merge them together: for each common team, we collect the associated data from both input factorisations. The resulting factorisation (shown partially)

$$\begin{aligned}&\langle\text{Barcelona}\rangle \times (\langle\text{Messi}\rangle \cup \langle\text{Villa}\rangle)\\ &\quad\times (\langle\text{Primera}\rangle \times \langle\text{CampNou}\rangle \cup\\ &\qquad\langle\text{Champions}\rangle \times (\langle\text{CampNou}\rangle \cup \langle\text{Wembley}\rangle))\\ &\quad\times \langle\text{Barcelona}\rangle \times \langle\text{CampNou}\rangle \cup \dots\end{aligned}$$

follows the f-tree $\mathcal{T}_5$ in Figure 3, which is obtained by merging the nodes team from $\mathcal{T}_1$ and $\mathcal{T}_4$. This join operator is called the *merge* operator and can be employed whenever the nodes to be joined are siblings or roots.

To execute the join on stadium, we must restructure again. One possibility is to swap each of the two nodes stadium in $\mathcal{T}_5$ with its parent, so that they become siblings (as in $\mathcal{T}_6$). In the factorisation, this amounts to regrouping the expressions under each team by stadium first. We can now execute the join using a merge operator on the nodes stadium (and obtain $\mathcal{T}_7$). In the factorisation, this means that for each team we merge the expressions grouped by stadiums, collecting the leagues and the city for each stadium:

$$\begin{aligned}&\langle\text{Barcelona}\rangle \times (\langle\text{Messi}\rangle \cup \langle\text{Villa}\rangle)\\ &\quad\times \langle\text{CampNou}\rangle \times (\langle\text{Primera}\rangle \cup \langle\text{Champions}\rangle)\\ &\quad\times \langle\text{Barcelona}\rangle \cup \dots,\end{aligned}$$

This completes the evaluation of Q_3. If we are only interested in the players from the same city as the stadium where they play, we further project on to $\{\text{stadium}, \text{player}\}$. The nodes league and city may be dropped straight away from $\mathcal{T}_7$, thereby obtaining the new f-tree $\mathcal{T}_8$ and the factorisation

$$\begin{aligned}&\langle\text{Barcelona}\rangle \times (\langle\text{Messi}\rangle \cup \langle\text{Villa}\rangle) \times \langle\text{CampNou}\rangle \cup\\ &\langle\text{Chelsea}\rangle \times (\langle\text{Cech}\rangle \cup \langle\text{Torres}\rangle) \times (\langle\text{Wembley}\rangle \cup \langle\text{Stamford}\rangle) \cup\\ &\langle\text{Arsenal}\rangle \times \langle\text{van Persie}\rangle \times (\langle\text{Wembley}\rangle \cup \langle\text{Stamford}\rangle).\end{aligned}$$

The node team cannot be dropped immediately. If (hypothetically) different teams had common players and played on the same stadiums, by removing this node we could obtain duplicate pairs of (player, stadium). In our scenario, we know that a player can only play for a team, i.e., there is a functional dependency from players to teams, and duplicates cannot occur. In the absence of this dependency, we must restructure $\mathcal{T}_8$ such that the node team becomes a leaf of the f-tree and thus there are distinct combinations of stadiums and players. For this, we apply two swap operators to first group by stadium ($\mathcal{T}_9$) and then by player ($\mathcal{T}_{10}$). We now drop the node team ($\mathcal{T}_{11}$) and obtain the final result

$$\begin{aligned}&\langle\text{CampNou}\rangle \times (\langle\text{Messi}\rangle \cup \langle\text{Villa}\rangle) \cup\\ &\langle\text{Wembley}\rangle \times (\langle\text{Cech}\rangle \cup \langle\text{Torres}\rangle \cup \langle\text{van Persie}\rangle) \cup\\ &\langle\text{Stamford}\rangle \times (\langle\text{Cech}\rangle \cup \langle\text{Torres}\rangle \cup \langle\text{van Persie}\rangle).\end{aligned}$$

Query Optimisation. The sequence of transformations from the input factorisation to the final factorised result can be captured by the swap, merge (join), and project operators. This factorisation plan can be schematically described in terms of transformations between f-trees, as shown in Figure 3. At each step of the plan, the structure given by the f-tree together with the input data define completely the factorisation of the intermediate result.

A query can have many factorisation plans. The space of plans is partly defined by the order of joins and, novel in FDB, by the different possible factorisation structures for the intermediate and final results. This space is inherently exponential and choosing a good plan is the key task of the FDB optimiser [3]. A good plan yields a small-size factorisation of the result and also has a small cost for the transformation of the input into the result factorisation. The two objectives may be contradictory since in order to achieve a minimal-size factorisation we may have to do more restructuring work. FDB currently supports two optimisation strategies: an exhaustive search strategy and a greedy

heuristic one. The former needs exponential time, whereas the latter only needs polynomial time in the size of the query. Preliminary experiments suggest that our heuristic produces plans very close to optimal ones while performing orders of magnitude faster than the exhaustive search. The quality of plans is assessed at compile time. For this, FDB computes the asymptotic complexity of the plans and of the size of their results, for any database. FDB can also use cardinality estimates and join selectivities, if available.

Let us consider the two plans for query Q_3 given in Figures 3 and 4. After first performing the join on team, the first plan restructures the factorisation such that the two nodes stadium become siblings, and then performs the join on stadium using the merge operator. An alternative plan could still join on team first (yielding the f-tree $\mathcal{T}_5$) but then swap one of the nodes stadium twice until it becomes the root, and then perform the join on stadium using an *absorb* operator. This join operator is only applicable if the nodes to be joined are in a ancestor-descendant relationship, as it is the case for the two nodes stadium in $\mathcal{T}_7'$. The join effectively merges the two nodes into the top one. In the factorisation, this operator filters out all expressions rooted at the lower node stadium where its value is different from the value at the ancestor node stadium. The fragment of this alternative second plan that differs from the first plan is shown in Figure 4.

IMDB and DBLP Scenarios. The IMDB (Internet Movie Database) data set consists of relations associating to each movie the actors and actresses playing in the movie, its directors, its genres, and associated keywords. It contains fewer but much larger relations than NELL (12M actors). Typical queries join these relations on movie to relate, e.g., actors and actresses involved in the same movie. This scenario will also illustrate queries with self-joins to relate actors that co-starred in movies with the same actress. We will use DBLP data, which associates authors, venues, and dates to publications, to answer questions about relationships of authors similar to actors in the IMDB scenario.

Crossword Scenario. This scenario uses crosswords to illustrate how FDB can manage the solutions to a set of constraints as it is gradually refined. A crossword with cells for letters arranged into a grid of intersecting words can be represented by a relational schema: each cell is represented by an attribute, each sequence of cells to be filled with a clue (i.e., word) is represented by a relation. For each clue, the corresponding relation contains the set of possible words to be filled into the clue, each word corresponds to a tuple of letters. The relations are joined whenever two clues intersect in a cell; the join of all relations represents the set of all possible solutions to the crossword, i.e., the set of possible assignments of letters to all attributes or cells of the grid.

Initially, the relations corresponding to clues contain English words of correct length. Upon obtaining additional information, such as revealing a letter or restricting a clue to a smaller class of words (e.g., city names), the user can apply further selections (equating an attribute to a constant letter) or joins (joining clues with lists of city names) to the solution set. Additionally, the user can ask for the current possibilities for a particular cell or clue via queries.

FDB is suited for managing such a database of crossword solutions that can be prohibitively large to store explicitly. A good factorisation structure (f-tree) can be inferred from the shape of the crossword grid alone.

3. USER INTERACTION WITH FDB

FDB System. FDB is an in-memory query engine implemented in C++. It has a query parser, an optimiser, and an evaluator. For cost estimation, it uses a system catalogue to store cardinality estimates and join selectivities, and a module for computing size bounds for factorised representations using linear programs solved using the GLPK package v4.45. Both factorised relations and f-trees are represented in memory as trees and can be serialised to (and read from) disk using the left-to-right depth-first preorder traversal. FDB can also perform rudimentary relational query evaluation of select-project-join queries using a multi-way merge-sort join followed by projections. Relational data can be read from and stored into CSV files.

FDB User Interface. The interface allows users to specify and execute select-project-join queries on factorised databases. For ease of demonstration, a set of predefined scenarios (data sets and queries) are readily available, as detailed in Section 2. Optionally, a choice of exhaustive or greedy optimisation strategy can be specified; the greedy strategy is used by default. Factorisation plans can be serialised to and read from disk, executed on a factorised database, and visualised using LaTeX as in Figures 3 and 4. Factorised results are visualised in tabular form or in XML format. For each step in a factorisation plan, it is possible to inspect its input and output factorisation trees and factorised relations. For the latter, the interface also displays their sizes, i.e., number of data elements, and the sizes of the equivalent flat relational representations, i.e., number of tuples times the arity of the relation. This shows the succinctness gap brought by factorisations. To demonstrate the time performance gap between FDB and relational engines, we will run on the same scenarios a home-bred in-memory relational engine [3] as well as PostgreSQL and SQLite.

4. REFERENCES

[1] L. Antova, C. Koch, and D. Olteanu. 10^{10^6} Worlds and Beyond: Efficient Representation and Processing of Incomplete Information. In *ICDE*, 2007.

[2] M. Aschinger, C. Drescher, G. Gottlob, P. Jeavons, and E. Thorstensen. Tackling the partner units configuration problem. In *IJCAI*, 2011.

[3] N. Bakibayev, D. Olteanu, and J. Závodný. FDB: A query engine for factorised relational databases. *PVLDB*, 5(12), 2012.

[4] A. Carlson, J. Betteridge, B. Kisiel, B. Settles, E. H. Jr., and T. Mitchell. Toward an architecture for Never-Ending Language Learning. In *AAAI*, 2010.

[5] T. Imielinski, S. Naqvi, and K. Vadaparty. Incomplete objects — a data model for design and planning applications. In *SIGMOD*, 1991.

[6] D. Olteanu and J. Závodný. On factorisation of provenance polynomials. In *TaPP*, 2011.

[7] D. Olteanu and J. Závodný. Factorised representations of query results: Size bounds and readability. In *ICDT*, 2012.

[8] D. Suciu, D. Olteanu, C. Ré, and C. Koch. *Probabilistic Databases*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2011.