On the Power of Database Update Languages

Michael Lawley

School of Computing and Information Technology
Griffith University
Nathan, Queensland, Australia 4111

Abstract

A database update may be considered a mapping from one database instance to another database instance. A database update language L_1 is considered more powerful than a database update language L_2 if L_1 can express a superset of the mappings expressible in L_2. We consider several database update languages described in the literature. We present them in a uniform notation with well defined semantics and examine their relative expressive power.

Keywords Database, update, expressive power

1 Introduction

We study the relative expressive power of several database update languages that have been described in the literature. In doing so, we attempt to classify their expressive power in the same way that Chandra[4] classifies the expressive power of database query languages. A similar classification was done by Abiteboul and Vianu[1,2] but we deal with a set of languages that is less expressive (and more practical) than theirs.

For any update language with alternation or iteration constructs, the expressive power of the language as a whole will be affected by the power of the language available for expressing the conditions in these constructs. We restrict our study to deterministic update languages, focusing on the

iteration constructs and how their power depends on the language available for expressing their conditions.

An understanding of the expressive power of update languages is essential for the efficient checking of integrity constraints. We intend to explore this application in subsequent work.

The paper is arranged as follows. Section 2 gives a model for database updates and introduces some necessary notation and concepts. Section 3 describes the semantics of several proposed update languages while Section 4 examines the relative power of these languages. Section 5 concludes with a summary of the results from Section 4 and suggestions for further research.

2 Preliminaries

Let $\mathcal{U}$ be a *universal domain* consisting of a countably infinite set of constants. A *database* is a tuple $\langle R_1, \ldots, R_n \rangle$, where each R_i is a finite named relation over $\mathcal{U}^{k_i}$ for some $k_i \geq 0$. A *database update* is a mapping from one database, $B = \langle R_1, \ldots, R_n \rangle$, to another, $B' = \langle R'_1, \ldots, R'_n \rangle$, where each R_i and R'_i have the same arity. We require the mapping to be partially recursive and C-generic for some finite set of constants C. (See Abiteboul and Vianu[2] for an explanation of the C-genericity condition.)

Some of the update languages we consider contain iteration and alternation constructs which depend on some query. The power of the language used for expressing this query will affect the power of the update language. This is very important since, if we cannot predict the result of a particular query, then we will have difficulty in reasoning about an update involving that query. Figure 1 shows the relative power of the query languages we consider.[4]

Let L be an update language and M a query language. We write $\mathrm{L_M}$ to refer to the update language with alternation and iteration conditions expressed in that query language; e.g., $\mathrm{L_{SPJ}}$ and $\mathrm{L_{FO}}$ refer to L with iteration and alternation conditions specified in the query languages SPJ and FO respectively.

Let $\mathrm{L_1}$ and $\mathrm{L_2}$ be update languages. $\mathrm{L_2}$ is at least as powerful as $\mathrm{L_1}$ if every mapping expressible in $\mathrm{L_1}$ is also expressible in $\mathrm{L_2}$. We write this as $\mathrm{L_1} \sqsubseteq \mathrm{L_2}$. If the set of mappings expressible in $\mathrm{L_2}$ is a strict superset of those expressible in $\mathrm{L_1}$, then $\mathrm{L_2}$ is more powerful than $\mathrm{L_1}$ and we write

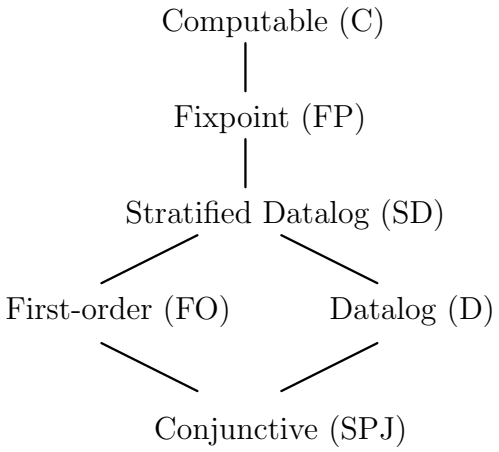

Figure 1: Expressive power of query languages.

$L_1 \sqsubset L_2$. If the set of mappings expressible in L_1 is not fully contained in the set of mappings expressible in L_2 and vice versa then the languages are incomparable and we have $L_1 \not\sqsubset L_2$ and $L_2 \not\sqsubset L_1$.

3 Some Update Languages

In this section we introduce a selection of update languages from the literature. We believe the selection covers most of the common update language features including the presence/absence of iteration; bounded/unbounded iteration; and general/restricted loop conditions. They are presented in a uniform syntax to make the differences and similarities between them more apparent.

In the following, let $\overline{a}$ and $\overline{c}$ be ground tuples, $F(\overline{x})$ a query in some language with free variables $\overline{x}$, $R(\overline{x})$ a named relation with unbound variables $\overline{x}$, and $S[\overline{x}]$ a statement in the update language with free variables $\overline{x}$.

3.1 LST – An Iteration free Language

In general, these consist of finite sequences of insertions and deletions.[5,7] They have the standard requirement that a sequence may not both insert and delete the same tuple, and thus the order of operations is unimportant. Figure 2 gives the formal syntax.

$$\begin{aligned} \mathit{stmt} ::= {} & \mathit{stmt}\ ;\ \mathit{stmt} \\ | \quad & \mathit{insert}_R(\overline{a}) \\ | \quad & \mathit{delete}_R(\overline{a}) \end{aligned}$$

Figure 2: Syntax of the iteration free language, LST.

3.2 SdetTL – An Iterator Based Language

Abiteboul and Vianu's language SdetTL[2] is given in Figure 3.

$$\begin{aligned} \mathit{stmt} ::= {} & \mathit{stmt}\ ;\ \mathit{stmt} \\ | \quad & \mathit{delete}_R(\overline{a}) \\ | \quad & \mathit{insert}_R(\overline{a}) \\ | \quad & \mathit{erase}(R) \\ | \quad & \mathit{while}\ \overline{x} : F(\overline{x})\ \mathit{do}\ \mathit{stmt} \end{aligned}$$

Figure 3: Syntax of the iterator based language, SdetTL.

The *while* construct requires a little explanation. Its semantics are such that, at each iteration, *stmt* is performed in parallel for every $\overline{x}$ satisfying $F(\overline{x})$. The result of each iteration is then the union of each parallel branch. The process is then repeated until $F(\overline{x})$ can no longer be satisfied or until *stmt* no longer has an effect (i.e. a fixpoint has been generated). This is termed *saturation*. It should be noted that these semantics require every branch to delete the same tuple for the deletion of that tuple to be effective, but only one branch need insert a tuple.

An alternative semantics, where *stmt* is performed for only one instantiation of $\overline{x}$ satisfying $F(\overline{x})$ at each iteration, results in non-determinism.

Example 3.1 Consider the update $\mathit{while}\ \overline{x} : R(\overline{x})\ \mathit{do}\ \mathit{delete}_R(\overline{x})$. With a deterministic semantics, one would expect this update to delete every tuple

in the relation R. However, since each parallel branch deletes a different tuple, the update has no effect on R. This means it is ineffective to put a *delete* in the loop.

On the other hand, with non-deterministic semantics, the update will delete every tuple in R in some, unspecified, order. □

Non-deterministic languages such as detTL and TL (also proposed by Abiteboul and Vianu[2]) are outside the scope of this paper.

3.3 WL – A Set Based Language

Wallace's proposed language[10] is given in Figure 4.

$$
\begin{aligned}
\mathit{stmt} ::= \; & \mathit{stmt} \; ; \; \mathit{stmt} \\
| \; & \mathit{insert}_R(\overline{a}) \\
| \; & \mathit{delete}_R(\overline{a}) \\
| \; & \mathit{replace}_R(\overline{a}, \overline{c}) \\
| \; & \mathit{if}\ F\ \mathit{then}\ \mathit{stmt} \\
| \; & \mathit{foreach}\ \overline{x} : F(\overline{x})\ \mathit{do}\ \mathit{stmt}
\end{aligned}
$$

Figure 4: Syntax of the set based language, WL.

The statement $\mathit{foreach}\ \overline{x} : F(\overline{x})\ \mathit{do}\ \mathit{stmt}$ is executed by evaluating $F(\overline{x})$ to produce a set of bindings for $\overline{x}$ and then executing *stmt* in the following manner. If *stmt* is an atomic statement ($\mathit{insert}_R(\overline{x})$, $\mathit{delete}_R(\overline{x})$ or $\mathit{replace}_R(\overline{x}, \overline{y},)$), then execute it for each $\overline{x}$ binding in parallel. If *stmt* is a sequence of statements, $\mathit{stmt}_1; \ldots ; \mathit{stmt}_n$, then execute stmt_1 for each $\overline{x}$ binding in parallel, then execute stmt_2 for each $\overline{x}$ binding in parallel, and so on. Otherwise, *stmt* is $\mathit{foreach}\ \overline{y} : F'(\overline{x}, \overline{y})\ \mathit{do}\ \mathit{stmt}'$, where $\overline{x}$ and $\overline{y}$ are disjoint. Evaluate $F'(\overline{x}, \overline{y})$ for each $\overline{x}$ binding to produce a new set of bindings for $\overline{x}$ and $\overline{y}$. Then execute stmt' for the new set of bindings as described above.

Note that the condition of an *if* statement has no free variables, and consequently is a special case of the *foreach* statement. The *while* statement is shorthand for a *delete* followed by an *insert*.

Example 3.2 If the query, $F(x, y)$, produces the set of bindings $\{\langle a, b\rangle, \langle b, a\rangle\}$, then the update:

$$\mathit{foreach}\ x, y : F(x, y)\ \mathit{do}\ (\mathit{insert}_R(x)\ ;\ \mathit{delete}_R(y))$$

is equivalent to

$$insert_R(a)\ ;\ insert_R(b)\ ;\ delete_R(b)\ ;\ delete_R(a)$$

and not

$$insert_R(a)\ ;\ delete_R(b)\ ;\ insert_R(b)\ ;\ delete_R(a)$$

□

The following result will allow us to ignore nested *foreach* statements when comparing WL with other languages.

Lemma 3.1 *Any nested foreach construct can be flattened to a sequence of simple foreach statements containing only primitive updates.*

Proof sketch: By rewriting *if*'s as *foreach*'s and repeatedly applying the following transformations, we produce an equivalent flat statement.

1. $$\begin{array}{l} foreach\ \overline{x} : F_1(\overline{x})\ do\ foreach\ \overline{y} : F_2(\overline{x},\overline{y})\ do\ \mathrm{S}[\overline{x},\overline{y}] \\ \qquad \Downarrow \\ foreach\ \overline{x},\overline{y} : F_1(\overline{x}) \wedge F_2(\overline{x},\overline{y})\ do\ \mathrm{S}[\overline{x},\overline{y}] \end{array}$$

2. Let R_T be a new empty relation with the appropriate arity.

$$\begin{array}{l} foreach\ \overline{x} : F(\overline{x})\ do\ (\mathrm{S}_1[\overline{x}]\ ;\ \ldots\ ;\ \mathrm{S}_n[\overline{x}]) \\ \qquad \Downarrow \\ foreach\ \overline{x} : F(\overline{x})\ do\ insert_{R_T}(\overline{x})\ ; \\ foreach\ \overline{x} : R_T(\overline{x})\ do\ \mathrm{S}_1[\overline{x}]\ ; \\ \qquad \vdots \\ foreach\ \overline{x} : R_T(\overline{x})\ do\ \mathrm{S}_n[\overline{x}] \end{array}$$

These transformation will always preserve the original semantics since, within the body of a *foreach*, each operation in a sequence is performed for the entire set of loop tuples before the next one in the sequence. Hence, the effect of inserting, deleting or replacing a tuple multiple times is the same as doing it just once.

3.4 SS – A Non-Orthogonal Language

Figure 5 gives the non-orthogonal language proposed by Stemple and Sheard.[9]

$$
\begin{aligned}
\mathit{stmt} ::= {} & \mathit{stmt}\ ;\ \mathit{stmt} \\
| \quad & \mathit{insert}_R(\overline{a}) \\
| \quad & \mathit{delete}_R(\overline{a}) \\
| \quad & \mathit{foreach}\ \overline{x} : F(\overline{x})\ \mathit{do}\ \mathit{delete}_R(\overline{x}) \\
| \quad & \mathit{foreach}\ \overline{x} : R_1(\overline{x})\ \mathit{do}\ \mathit{insert}_{R_2}(f(\overline{x})) \\
| \quad & \mathit{foreach}\ \overline{x} : R(\overline{x}) \wedge F(\overline{x})\ \mathit{do}\ \mathit{replace}_R(\overline{x}, f(\overline{x})) \\
| \quad & \mathit{foreach}\ \overline{x} : R(\overline{x}) \wedge F_1(\overline{x})\ \mathit{do} \\
& \quad \mathit{if}\ F_2\ \mathit{then}\ \mathit{delete}_R(\overline{x})\ \mathit{else}\ \mathit{replace}_R(\overline{x}, f(\overline{x})) \\
| \quad & \mathit{if}\ F\ \mathit{then}\ \mathit{stmt}\ \mathit{else}\ \mathit{stmt}
\end{aligned}
$$

Figure 5: Syntax of the non-orthogonal language, SS.

Here, $f(\overline{x})$ is an arbitrary function which takes the input tuple $\overline{x}$ and returns a tuple of appropriate type for the relation.

It can be seen that this is almost a subset of Wallace's language. The only difference is the *if-then-else* statement which we deal with in Section 4.

3.5 T – A "Query Based" Update Language

Abiteboul and Vianu[2] describe updates and queries as being instances of "database transformations" in the sense that they map database instances to other database instances. This idea is also presented by Qian.[8] We now introduce the "T" family of update languages, allowing us to treat query languages as update languages.

For each relation R_i to be altered by an update, let $F_i(\overline{x})$ be a safe query such that the updated relation R'_i is described by the set $\{\overline{x} \mid F_i(\overline{x})\}$. So, for the update language $\mathrm{T}_{\mathrm{SPJ}}$, all the queries $F_i(\overline{x})$ are formulae in SPJ, and similarly for T_{FO}, T_{D}, etc.

We justify the restriction to safe queries by observing that updates can only operate on existing tuples, and thus are inherently range restricted.

Example 3.3 In $\mathrm{T}_{\mathrm{SPJ}}$, to insert the intersection of the relations R_2 and R_3 into R_1 we write

$$R_1 \leftarrow \{x, y \mid R_2(x, y) \wedge R_3(x, y)\}$$

□

Example 3.4 In $\mathrm{T_{FO}}$, to delete all paths starting at m from the path relation R we write

$$R \leftarrow \{x, y \mid R(x, y) \wedge \neg(x = m)\}$$

This is neither a valid $\mathrm{T_{SPJ}}$ nor $\mathrm{T_D}$ update because it contains negation. □

The purpose here is to get a feel for the power of the various update language constructs when combined with conditions expressed in each of the above query languages. Note that the relative power of $\mathrm{T_{SPJ}}$, $\mathrm{T_{FO}}$, $\mathrm{T_D}$, etc. is the same as that of the query languages illustrated in Figure 1.

4 Relative Power

We first consider results that are independent of the query language used. We then examine the effect of the query language on the power of the update language.

4.1 General Results

Lemma 4.1 *LST is strictly less powerful than SS (LST $\sqsubset$ SS).*

Proof sketch: LST does not contain any form of looping construct so an update which is to operate on every tuple in a relation cannot be independent of the current state of that relation. For example, LST cannot delete every tuple from a relation.

Lemma 4.2 *SS is strictly less powerful than WL (SS $\sqsubset$ WL).*

Proof sketch: First, any SS update can be rewritten in WL. We need only consider *if-then-else* which can be rewritten as follows

$$\begin{array}{l} \textit{if } F \textit{ then } \textit{stmt}_1 \textit{ else } \textit{stmt}_2 \\ \qquad \Downarrow \\ \textit{insert}_{R_T}(\textit{else}) \\ \textit{if } F \textit{ then } (\ \textit{delete}_{R_T}(\textit{else})\ ;\ \textit{stmt}_1\ ) \\ \textit{if } R_T(\textit{else}) \textit{ then } (\ \textit{delete}_{R_T}(\textit{else})\ ;\ \textit{stmt}_2\ ) \end{array}$$

where R_T is a new, initially empty, unary relation. Nested *if*s will also work since the relation R_T is always empty before $stmt_i$ is performed and is left empty no matter which branch is taken.

For WL, we note that, when nested inside a *foreach*, the condition is evaluated for each variable binding in parallel, followed by all the *then* statements in parallel, followed by all the *else* statements in parallel since, as translated above, *if-then-else* consists of two sequential *if* statements. This is a natural semantics for WL, since the alternative of having the *then* and *else* branches performed in parallel introduces the possibility of non-determinism. Because of the constrained syntax of SS updates, the issue of the order of updates does not arise.

Second, there is no way to generate a "product" in SS. This is because the iterative constructs of SS may only range over a single relation and there is no way to nest these constructs.

Lemma 4.3 *WL is less powerful than SdetTL when the query language contains negation (WL $\sqsubseteq$ SdetTL) and they are incomparable otherwise. The exact relationship between WL and SdetTL depends on the query language and is given below.*

Proof sketch: Any WL update can be rewritten as an SdetTL update. To do this we must be able to express *foreach* as a *while*. The mapping

$$
\begin{array}{l}
\mathit{foreach}\ \overline{x} : F(\overline{x})\ \mathit{do}\ \mathit{stmt} \\
\qquad \Downarrow \\
\mathit{insert}_{R_T}(a) \\
\mathit{while}\ \overline{x} : F(\overline{x}) \wedge R_T(a)\ \mathit{do}\ (\mathit{delete}_{R_T}(a)\ ;\ \mathit{stmt})
\end{array}
$$

is incorrect since *stmt* may include a delete and, due to the semantics of the *while* requiring the delete to be performed in every parallel branch, this effectively nullifies the deletion. However, deletes can be implemented as follows:

- replace anything of the form $\mathit{delete}_R(\overline{a})$ by $\mathit{insert}_{R'}(\overline{a})$,
- replace all the corresponding references $R(\overline{x})$ by $(R(\overline{x}) \wedge \neg R'(\overline{x}))$.
- At the end of the update, copy all tuples in R and not in R' into another temporary relation R'', erase R, and copy R'' into R.

Example 4.1 The following update

$$\mathit{foreach}\ \overline{x} : F(\overline{x})\ \mathit{do}\ \mathit{delete}_R(\overline{x})$$

becomes

$$\begin{array}{l}\mathit{insert}_{R_T}(a)\\ \mathit{while}\ \overline{x} : F(\overline{x}) \wedge R_T(a)\ \mathit{do}\ (\mathit{delete}_{R_T}(a)\ ;\ \mathit{insert}_{R'}(\overline{x}))\\ \mathit{while}\ \overline{x} : R(\overline{x}) \wedge \neg R'(\overline{x})\ \mathit{do}\ \mathit{insert}_{R''}(\overline{x})\\ \mathit{erase}(R)\\ \mathit{while}\ \overline{x} : R''(\overline{x}) \wedge \neg R(\overline{x})\ \mathit{do}\ \mathit{insert}_R(\overline{x})\end{array}$$

Also, any reference $R(\overline{x})$ in $F(\overline{x})$ must be mapped to $R(\overline{x}) \wedge \neg R'(\overline{x})$. □

In summary, these results give us LST $\sqsubset$ SS $\sqsubset$ WL $\sqsubseteq$ SdetTL, the last inclusion only holding when the query language contains negation.

4.2 Query language specific results

We now examine how the power of the query language affects the relative expressive power of WL and SdetTL. We choose to omit LST and SS from this discussion since both WL and SdetTL are more general.

Lemma 4.4 *T_{SPJ} is strictly less powerful than WL_{SPJ} ($T_{\mathrm{SPJ}} \sqsubset WL_{\mathrm{SPJ}}$).*

Proof sketch: The *foreach* of WL allows us to express any T update as follows:

$$\begin{array}{l}R \leftarrow \{\overline{x} \mid F(\overline{x})\}\\ \qquad \Downarrow\\ \mathit{foreach}\ \overline{x} : F(\overline{x})\ \mathit{do}\ \mathit{insert}_R(\overline{x})\end{array}$$

However, $\mathrm{T}_{\mathrm{SPJ}}$ cannot express the $\mathrm{WL}_{\mathrm{SPJ}}$ update $\mathit{insert}_R(a)$, since this requires an increase in the cardinality of the relation R, which is not possible with only *select*, *project*, and *join*.

Lemma 4.5 *T_{SPJ} is strictly less powerful than $SdetTL_{\mathrm{SPJ}}$ ($T_{\mathrm{SPJ}} \sqsubset SdetTL_{\mathrm{SPJ}}$).*

Proof sketch: Any T update can be rewritten as an SdetTL update as given above but with *while* implementing *foreach*. $\mathrm{T}_{\mathrm{SPJ}}$ cannot express transitive closure[3] while $\mathrm{SdetTL}_{\mathrm{SPJ}}$ can, as follows:

$$\textit{while } x, y, z : R(x, y) \wedge R(y, z) \textit{ do } \textit{insert}_R(x, z)$$

Lemma 4.6 *$SdetTL_{\text{SPJ}}$ is incomparable with WL_{SPJ}.*

Proof sketch: First, $\text{SdetTL}_{\text{SPJ}}$ can express updates that WL_{SPJ} cannot. As outlined above, any WL update can be flattened into a (finite) sequence of unnested *foreach* statements. Since the transitive closure of a relation cannot be expressed as an SPJ query, any WL_{SPJ} update can only extend the relation by a fixed number of paths. $\text{SdetTL}_{\text{SPJ}}$ as shown above, can express the transitive closure of a relation.

Conversely, WL_{SPJ} can express updates that $\text{SdetTL}_{\text{SPJ}}$ cannot. Consider the following WL_{SPJ} update:

$$\textit{foreach } \overline{x} : R_1(\overline{x}) \textit{ do } \textit{delete}_{R_2}(\overline{x})$$

Because of the union-parallel semantics of *while*, the only way to reduce the size of a relation is to erase it then add tuples back. The tuples to be added back are exactly $R_2 \wedge \neg R_1$ which is not expressible with an SPJ query. Hence, this update is not expressible in $\text{SdetTL}_{\text{SPJ}}$.

Lemma 4.7 *WL_{SPJ} and T_{FO} are equivalent.*

Proof sketch: To convert from T_{FO} to WL_{SPJ}, convert the FO formula to existential normal form (writing each $\forall$ as $\neg\exists\neg$). Negation can then be implemented with *delete*.

Example 4.2 The following update

$$R \leftarrow \{z \mid R_1(z) \wedge \forall x \ (R_2(x, z) \rightarrow R_3(x))\}$$

is written as

$$R \leftarrow \{z \mid R_1(z) \wedge \neg\exists x \ (R_2(x, z) \wedge \neg R_3(x))\}$$

which translates to

$$\begin{array}{l}
\textit{foreach } x, z : R_2(x, z) \textit{ do } \textit{insert}_{R_T}(x, z) \\
\textit{foreach } x, z : R_3(x) \wedge R_T(x, z) \textit{ do } \textit{delete}_{R_T}(x, z) \\
\textit{foreach } x : R(x) \textit{ do } \textit{delete}_R(x) \\
\textit{foreach } z : R_1(z) \textit{ do } \textit{insert}_R(z) \\
\textit{foreach } x, z : R_T(x, z) \textit{ do } \textit{delete}_R(z)
\end{array}$$

□

To go the other way, from $\mathrm{WL}_{\mathrm{SPJ}}$ to T_{FO}, flatten the $\mathrm{WL}_{\mathrm{SPJ}}$ update as described in Lemma 3.1; each *foreach* can then be rewritten as an FO statement as follows.

$$\begin{array}{l} \textit{foreach } \overline{x} : F(\overline{x}) \textit{ do } \textit{insert}_R(\overline{x}) \\ \qquad \Downarrow \\ R \leftarrow \{\overline{x} \mid R(\overline{x}) \vee F(\overline{x})\} \end{array}$$

$$\begin{array}{l} \textit{foreach } \overline{x} : F(\overline{x}) \textit{ do } \textit{delete}_R(\overline{x}) \\ \qquad \Downarrow \\ R \leftarrow \{\overline{x} \mid R(\overline{x}) \wedge \neg F(\overline{x})\} \end{array}$$

$$\begin{array}{l} \textit{foreach } \overline{x}, \overline{y} : F(\overline{x}, \overline{y}) \textit{ do } \textit{replace}_R(\overline{x}, \overline{y}) \\ \qquad \Downarrow \\ R \leftarrow \{\overline{x} \mid (R(\overline{x}) \wedge \neg F(\overline{x}, \overline{y})) \vee (R(\overline{y}) \wedge F(\overline{y}, \overline{x}))\} \end{array}$$

Lemma 4.8 *T_{D} is strictly less powerful than $\mathrm{SdetTL}_{\mathrm{SPJ}}$ ($\mathrm{T}_{\mathrm{D}} \sqsubset \mathrm{SdetTL}_{\mathrm{SPJ}}$).*
Proof sketch: T_{D} cannot express the $\mathrm{SdetTL}_{\mathrm{SPJ}}$ update $\textit{delete}_R(\overline{a})$. A T_{D} update involving a rule of the form

$$R_1(\overline{x}) \leftarrow R_2(\overline{y}) \wedge \cdots \wedge R_1(\overline{z})$$

can be rewritten in $\mathrm{SdetTL}_{\mathrm{SPJ}}$ as

$$\textit{while } \overline{x} : R_2(\overline{y}) \wedge \cdots \wedge R_1(\overline{z}) \textit{ do } \textit{insert}_{R_1}(\overline{x})$$

where $\overline{x}$, $\overline{y}$, and $\overline{z}$ are not necessarily disjoint.

Lemma 4.9 *$\mathrm{SdetTL}_{\mathrm{SPJ}}$ is strictly less powerful than T_{SD} ($\mathrm{SdetTL}_{\mathrm{SPJ}} \sqsubset \mathrm{T}_{\mathrm{SD}}$).*
Proof sketch: $\mathrm{SdetTL}_{\mathrm{SPJ}}$ cannot delete sets of tuples due to the semantics of the *while* statement (see Example 3.1). T_{SD} can, however, effectively delete tuples using negation. $\mathrm{SdetTL}_{\mathrm{SPJ}}$ updates can be rewritten in T_{SD} in a similar manner to Lemma 4.8.

Lemma 4.10 *$\mathrm{SdetTL}_{\mathrm{D}}$ is equivalent to $\mathrm{SdetTL}_{\mathrm{SPJ}}$.*
Proof sketch: The difference between D queries and SPJ queries is due to the presence of recursion, which is already (effectively) present in $\mathrm{SdetTL}_{\mathrm{SPJ}}$ in the form of the *while* statement.

Lemma 4.11 *WL_{FO}, WL_{SPJ} and T_{FO} are all equivalent in power.*

Proof sketch: The difference between SPJ queries and FO queries is due to the presence of disjunction and negation. These can already be simulated by sequence and deletion in WL_{SPJ}, so they add no power to the language WL_{FO}. The translation of WL_{FO} updates to T_{FO} updates and vice versa is trivial.

Lemma 4.12 *WL_{D}, WL_{SD}, and T_{SD} are all equivalent in power.*

Proof sketch: The negation present in WL_{SD} can be implemented by deletion in WL_{D}. The translation of WL_{SD} updates to T_{SD} updates and vice versa is trivial.

Lemma 4.13 *$SdetTL_{\text{FO}}$ and T_{FP} are equivalent in power.*

Proof sketch: $\text{SdetTL}_{\text{FO}}$ has negation, so can effectively perform deletes. The *while* statement provides a fixpoint operator. This means it can, for example, express *win*,[6] which is the standard example of a query which can be written in FP and not SD.

Lemma 4.14 *WL_{FP}, $SdetTL_{\text{SD}}$, $SdetTL_{\text{FP}}$, and T_{FP} are all equivalent in power.*

Proof sketch: The *while* statement of $\text{SdetTL}_{\text{SD}}$ can be used to implement the FP queries of WL_{FP}, $\text{SdetTL}_{\text{FP}}$, and T_{FP}. Similarly, FP queries in WL_{FP} and T_{FP} can be used to implement the *while* statement of $\text{SdetTL}_{\text{SD}}$ and $\text{SdetTL}_{\text{FP}}$.

Lemma 4.15 *WL_{C}, $SdetTL_{\text{C}}$, and T_{C} are all equivalent in power.*

This is self-evident.

4.3 Summary of Results

The major observations to be made here are the relationships between the update language constructs (of WL and SdetTL) and the query language constructs (of T). We note that deletions can effectively be performed by inserting the difference of two sets of tuples using negation.

Figure 6 summarises the results presented in this section.

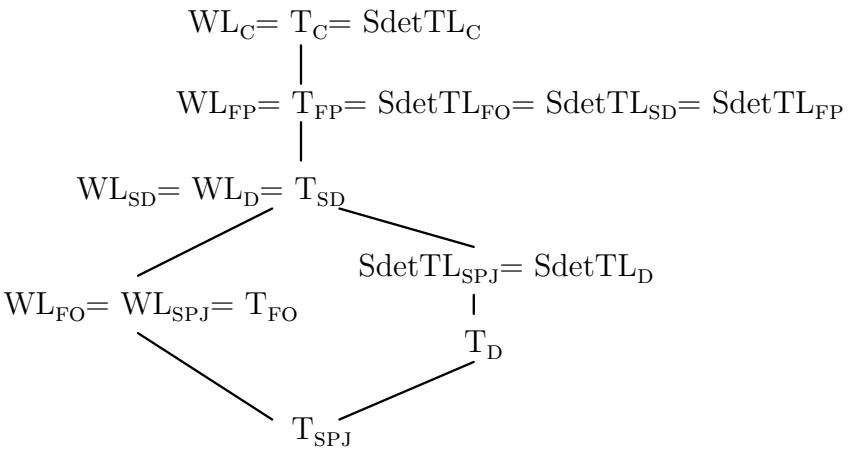

Figure 6: Expressive power of update languages.

5 Conclusion and Future Work

We have shown how the relative power of several update languages can be characterised and have provided a comparison with update languages closely related to a well known set of query languages. These results give a good indication of the difficulty in reasoning about the effects of updates in various languages and provide a basis for examining the relationship between integrity constraints and weakest preconditions.

On a more practical level, we still need to investigate the effect of introducing aggregate functions and arithmetic to the update languages. These capabilities are necessary in any actual implementation of a database update language.

Acknowledgement

Many thanks to Rodney Topor for valuable input on the topic of this paper and to Gill Dobbie for reading and commenting on earlier drafts.

References

[1] S. Abiteboul and V. Vianu. Procedural and declarative database update languages. In *Proc. Seventh ACM Symposium on Principles of Database Systems*, pages 240–250, Austin, Texas, Mar. 1988.

[2] S. Abiteboul and V. Vianu. Procedural languages for database queries and updates. *Journal of Computer and System Sciences*, 41(1):181–229, 1990.

[3] A. V. Aho and J. D. Ullman. Universality of data retrieval languages. In *Conference Record of the Sixth ACM Symposium on Principles of Programming Languages*, pages 110–120, San Antonio, Texas, Jan. 1979.

[4] A. K. Chandra. Theory of database queries (extended abstract). In *Proc. Seventh ACM Symposium on Principles of Database Systems*, pages 1–9, Austin, Texas, Mar. 1988.

[5] L. J. Henschen, W. W. McCune, and S. A. Naqvi. Compiling constraint-checking programs from first-order formulas. In H. Gallaire, J. Minker, and J.-M. Nicolas, editors, *Advances in Data Base Theory*, pages 145–169. Plenum Press, New York, 1984.

[6] P. Kolaitis. The expressive power of stratified logic programs. manuscript, Department of Computer Science, Stanford University, 1987. To appear in *Information and Computation*.

[7] J. W. Lloyd, E. A. Sonenberg, and R. W. Topor. Integrity constraint checking in stratified databases. *Journal of Logic Programming*, 4(4):331–343, Dec. 1987.

[8] X. Qian. *The Deductive Synthesis of Database Transactions*. PhD thesis, Stanford University, Nov. 1989.

[9] T. Sheard and D. Stemple. Automatic verification of database transaction safety. *ACM Transactions on Database Systems*, 14(3):322–368, Sept. 1989.

[10] M. Wallace. Compiling integrity checking into update procedures. In *Proc. Twelfth International Joint Conference on Artificial Intelligence*, pages 903–908, Aug. 1991.