Accelerating Queries with Group-By and Join by Groupjoin

Guido Moerkotte
Universität Mannheim
Mannheim, Germany
moerkotte@informatik.uni-mannheim.de

Thomas Neumann
Technische Universität München
Munich, Germany
neumann@in.tum.de

ABSTRACT

Most aggregation queries contain both group-by and join operators, and spend a significant amount of time evaluating these two expensive operators. Merging them into one operator (the *groupjoin*) significantly speeds up query execution.

We introduce two main equivalences to allow for the merging and prove their correctness. Furthermore, we show experimentally that these equivalences can significantly speed up TPC-H.

1. INTRODUCTION

Most aggregation queries contain joins. For these queries it is almost inevitable to feature a join followed by a grouping with an aggregation. In a hash-based implementation, this results in a cascade of two hash tables: First, a hash table is built and maintained to compute the join result, and then, a second hash table is filled to compute the result of the aggregation.

It is easy to see that in some cases (i.e., when the group-by attributes and the join attributes are the same) it is sufficient to maintain only one hash table (see, e.g., [11]):

```
select   a,count(*)
from     R1 left outer join R2 on R1.a = R2.b
where    R1.c=5
group by a
```

Here, instead of producing join results immediately, the hash table can be used to collect and aggregate all join partners. While Klug concentrated on a special evaluation technique in which an index-nested loop join is used [11], we will present equivalences which allow to combine the group-by and the join into one *groupjoin* operator. Before we go any further, let us sketch the history of the groupjoin operator.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Articles from this volume were invited to present their results at The 37th International Conference on Very Large Data Bases, August 29th - September 3rd 2011, Seattle, Washington.
Proceedings of the VLDB Endowment, Vol. 4, No. 11
Copyright 2011 VLDB Endowment 2150-8097/11/08... $ 10.00.

The groupjoin is more than 20 years old. To the best of our knowledge, von Bültzingsloewen invented the groupjoin [19]. He named it *outer aggregation*. Thereafter, several new names were invented. For example, Nakano used the name *general aggregate formation* [16]. The reason might be that grouping is called *aggregate formation* by Klug [12]. Steenhagen, Apers, and Blanken introduced the name *nestjoin* [17]. Cluet and Moerkotte called it *binary grouping* [8]. Lately, Chatziantoniou, Akinde, Johnson, and Kim introduced the name *MD-Join* [5].

The groupjoin is quite versatile, and we strongly believe that no DBMS can do without it. For example, it has been successfully applied to the problem of unnesting nested queries in the context of SQL [2, 3, 16, 19, 20], OQL [6, 7, 8], and XQuery [13]. Chatziantoniou, Akinde, Johnson, and Kim apply the groupjoin to efficiently evaluate data warehouse queries which feature a `cube-by` or `group-by grouping sets` clause [5]. They translate these clauses into groupjoins.

As can be seen, previous work concentrated to exploit the groupjoin to decorrelate nested queries or evaluate advanced grouping clauses. In this paper, we study the problem of introducing groupjoins into aggregation queries in a systematic way. Our main contribution consists of two equivalences. The first equivalence allows to replace a sequence consisting of a left outerjoin and a grouping by a groupjoin. The second equivalence replaces a join followed by a grouping by a groupjoin. Looking at these equivalences, it should become clear why we prefer the name groupjoin. Although these equivalences look simple and are (quite) intuitive, their proofs are—compared to the proofs of other equivalences—rather complex and lengthy. We modularize the proof as much as possible, to have small useful pieces for other, similar proofs. Some simpler modules (which take of course the form of equivalences) will be presented in the main part of the paper. The proof itself, as well as an important lemma, is presented in the appendix. However, the reader is advised to read the proof, since it shows how the different pieces of the puzzle presented in the main body of the paper fall into their places to give the whole picture.

The rest of the paper is organized as follows. In Section 2, we review some basic definitions for aggregation functions and the definitions of the left outerjoin, grouping, and the groupjoin. Section 3 starts with simple equivalences for the group operator and the groupjoin. Then, the main equivalences are presented. In Section 4, we show how to apply the equivalences to some sample queries from TPC-H. Section 5 shows the runtimes for those queries to which our equivalences can be applied. Section 6 concludes the paper.

2. PRELIMINARIES

2.1 Notations

For some expression e which evaluates to a tuple or relation, we denote by $\mathcal{A}(e)$ the set of attributes e provides. By $\mathcal{F}(e)$ we denote the set of free (unbounded) attributes of e. For example, the attribute A occurs free in the selection predicate $A = 7$. By $\{\cdot\}_s$ we denote a set and by $\{\cdot\}_b$ a bag (multiset). Π denotes projection and Π^D denotes the duplicate-eliminating projection. $\chi_{a:e_2}(e_1)$ denotes the map operator, which evaluates an expression e_2 for every input tuple and stores the result in some new attribute a. $\triangleright$ denotes the antijoin, i.e., $e_1 \triangleright_q e_2$ contains all those tuples from e_1 which do not have a join partner in e_2.

In SQL, comparing two null values with a regular comparison operator (e.g., $=$) returns unknown. For the group operator, we need special comparison functions which return true for null equal null. We denote these comparisons by stacking a dot $(\cdot)$ on top of them, e.g., the equality predicate then becomes $\doteq$.

The identity function is denoted by id. We often denote null by $\perp$. Let A be a set of attributes, then $\perp_A$ denotes the tuple in attributes A with all their values being null. By TID we denote the tuple identifier of a tuple in some relation. We do not demand that it explicitly exists. We only need it conceptually. For some set of attributes A, we will use the functional dependency $A \rightarrow \text{TID}(e)$ to denote the fact that a tuple in some relation e is uniquely determined by the attributes contained in A. This implies that the relation is duplicate-free, which is the main purpose of this notation. This trick was also applied by Yan and Larson [21, 22], where they used RowId instead of TID.

2.2 Aggregation Functions

A scalar aggregation function $\text{agg} : \{\tau\}_b \rightarrow \mathcal{N}$ is called *decomposable* [9] if there exist functions

$$\begin{aligned} \text{agg}^1 : \{\tau\}_b &\rightarrow \mathcal{N}' \\ \text{agg}^2 : \{\mathcal{N}'\}_b &\rightarrow \mathcal{N} \end{aligned}$$

with

$$\text{agg}(Z) = \text{agg}^2(\{\text{agg}^1(X), \text{agg}^1(Y)\}_b)$$

for all X and Y (not empty) with $Z = X \cup Y$. This condition assures that $\text{agg}(Z)$ can be computed on arbitrary subsets (-lists, -bags) of Z independently, and the (partial) results can be aggregated to yield the correct total result. If the condition holds, we say that agg *is decomposable* with *inner* agg^1 and *outer* agg^2.

Table 1 summarizes the decomposition of well-known SQL aggregation functions. There, we denote by count^{NN} the aggregation function that counts the number of values which are not null. An expression of the form $\text{avg}(a)$ is evaluated by first simultaneously calculating the $\text{sum}(a)$ and $\text{count}^{\text{NN}}(a)$ and subsequently dividing the former by the latter.

We now extend the notion of decomposability to aggregation vectors. An aggregation vector is an expression of the form

$$(b_1 : \text{agg}_1(a_1), \ldots, b_k : \text{agg}_k(a_k))$$

where the a_i and b_i are attribute names and the agg_i are aggregation functions. Often, we will leave out the enclosing parenthesis and simply write

$$b_1 : \text{agg}_1(a_1), \ldots, b_k : \text{agg}_k(a_k).$$

agg	agg^1	agg^2
min	min	min
max	max	max
$\text{count}(*)$	$\text{count}(*)$	sum
$\text{count}(a)$	$\text{count}(a)$	sum
sum	sum	sum
avg	$\text{sum}, \text{count}^{\text{NN}}$	sum, sum

Figure 1: Decomposition of aggregate functions.

We use $\circ$ to denote the concatenation of two aggregation vectors.

Let $F = (b_1 : \text{agg}_1(a_1), \ldots, b_k : \text{agg}_k(a_k))$ be an aggregation vector and all aggregates agg_i be decomposable into agg_i^1 and agg_i^2. Then, we say that F is decomposable into F^1 and F^2 where

$$\begin{aligned} F^1 &:= (b'_1 : \text{agg}_1^1(a_1), \ldots, b'_k : \text{agg}_k^1(a_k)) \\ F^2 &:= (b_1 : \text{agg}_1^2(b'_1), \ldots, b_k : \text{agg}_k^2(b'_k)). \end{aligned}$$

Note that in all cases, if F is decomposable into F^1 and F^2, then F^1 is decomposable into $F^{1,1}$ and $F^{1,2}$, and F^2 is decomposable into $F^{2,1}$ and $F^{2,2}$. Further, we have

$$\begin{aligned} F^{1,1} &= F^1 \\ F^{1,2} &= F^2 \\ F^{2,1} &= F^2 \\ F^{2,1} &= F^2 \end{aligned}$$

Let e_1 and e_2 be arbitrary expressions. We say that an aggregation vector F is *splittable* into F_1 and F_2 with respect to e_1 and e_2 if $F = F_1 \circ F_2$, $\mathcal{F}(F_1) \cap \mathcal{A}(e_2) = \emptyset$, and $\mathcal{F}(F_2) \cap \mathcal{A}(e_1) = \emptyset$ [22]. Assume that F contains an aggregation function agg_i applied to some attribute a_i. If $a \in \mathcal{A}(e_1)$, then clearly $\text{agg}_i(a_i)$ belongs to F_1; if $a \in \mathcal{A}(e_2)$, then $\text{agg}_i(a_i)$ belongs to F_2. There are other cases where F is splittable. Consider, for example, $\text{sum}(a_1 + a_2)$ for $a_i \in \mathcal{A}(e_i)$. Since $\text{sum}(a_1 + a_2) = \text{sum}(a_1) + \text{sum}(a_2)$, this does not hinder splittability. The same holds for subtraction. Decomposability and splittability are both prerequisites for our main equivalences.

The correct handling of duplicates, i.e., bags, is essential for the correctness of the query compiler and requires some care. We will therefore classify our aggregation functions into those which are sensitive to duplicates and those which are not. An aggregation function is called *duplicate agnostic* if the multiplicity of the elements in the bag does not influence its result. It is called *duplicate sensitive* otherwise. For our aggregation functions we have

- min, max, sum(distinct), count(distinct), avg(distinct) are duplicate agnostic and
- sum, count, avg are duplicate sensitive.

Yan and Larson used the term Class C aggregation function for duplicate sensitive aggregation functions and Class D for duplicate agnostic aggregation functions [22].

The following definition also goes back to Yan and Larson [22]. Let $F = (b_1 : \text{agg}_1(a_1), \ldots, b_m : \text{agg}_m(a_m))$ be an aggregation vector. We define $F \otimes c$ for some attribute c, which will typically contain the result of some $\text{count}(*)$, as

$F \otimes c = (b_1 : \text{agg}'_1(e_1), \ldots, b_m : \text{agg}'_m(e_m))$ with

$$\text{agg}'_i(e_i) = \begin{cases} \text{agg}_i(e_i) & \text{if } \text{agg}_i \text{ is duplicate agnostic} \\ \text{agg}_i(e_i * c) & \text{if } \text{agg}_i(e_i) = \text{sum}(e_i) \\ \text{sum}(c) & \text{if } \text{agg}_i(e_i) = \text{count}(*) \end{cases}$$

Additionally, we define

$$\text{agg}'_i(\text{count}(e_i)) = \text{sum}(\texttt{if } e_i = \bot \texttt{ then } 0 \texttt{ else } c)$$

if $e_i \neq' *'$.

Finally, note that for all aggregate functions except count$(*)$, we have $\text{agg}(\{a\}) = a$ for arbitrary elements a. Thus, if we are sure that we deal with only one tuple, we can apply the following rewrite. Let a_i and b_i be attributes. Then, if $F = (b_1 : \text{agg}_1(a_1), \ldots, b_m : \text{agg}_m(a_m))$, we define $\hat{F} = (b_1 : a_1, \ldots, b_m : a_m)$.

2.3 Left Outerjoin

Although we assume the reader to be familiar with the left outerjoin, we give its definition here. However, while doing so, we slightly extend it to allow the left outerjoin to assign values other than NULL to some of the attributes of the right-hand side in case some tuple in its left argument does not find a join partner. We call these values *default values* and this extended version *left outerjoin with defaults*. Let $D = d_1 : c_1, \ldots, d_k : c_k$ be a vector assigning constants c_j to attributes d_j^i. We then define

$$\begin{aligned} e_1 \mathbin{⟕}^D_p e_2 \quad := \quad & (e_1 \mathbin{⋈}_p e_2) \\ & \cup((e_1 \mathbin{▷}_p e_2) \times \{\bot_{\mathcal{A}(e_2)\setminus\mathcal{A}(D)} \circ [D]\}). \end{aligned}$$

2.4 Group Operator

Grouping is defined on a bag, and its subscripts indicate (i) the grouping critera and (ii) a new attribute name as well as a function which is used to calculate its value.

$$\begin{aligned} \Gamma_{\theta G;g:f}(e) \quad := \quad & \{y \circ [g : x] \mid y \in \Pi^D_G(e), \\ & x = f(\{z | z \in e,\ z.G\ \theta\ y.G\}_b)\}_s \end{aligned}$$

for some set of attributes G, an attribute g and a function f. The comparison operator θ must be a null extended comparison operator like '$\dot{=}$'.

The grouping criterion may be defined on several attributes. Then, G and θ represent sequences of attributes and comparators. In case all θ equal '$\dot{=}$', we abbreviate $\Gamma_{\dot{=}G;g:f}$ by $\Gamma_{G;g:f}$.

We can extend the above definition to calculate several new attribute values by defining

$$\begin{aligned} \Gamma_{\theta G;b_1:f_1,\ldots,b_k:f_k}(e) := \{y \circ [b_1 : x_1, \ldots, b_k : x_k] \mid y \in \Pi^D_G(e), \\ x_i = f_i(\{z | z \in e,\ z.G\ \theta\ y.G\}_b)\}_s. \end{aligned}$$

We also introduce two variants of the grouping operator which can be used to abbreviate expressions. Let $F = b_1 : e_1, \ldots, b_k : e_k$ and $\mathcal{F}(e_i) = \{g\}$ for all $i = 1, \ldots, k$. Then we define

$$\Gamma_{G;F}(e) := \Pi_{\overline{g}}(\chi_F(\Gamma_{G;g:\text{id}}(e))).$$

Here, the free attribute g is implicit and $\Pi_{\overline{g}}$ removes it. If we wish to make it explicity, we write $\Gamma_{G;g;F}$ instead of simply $\Gamma_{G;F}$. Note that g plays the same role as `partition` in OQL ([4, p. 114]).

Let us also introduce an SQL-notation based variant, which we will use for the rest of the paper. Let F be an aggregation vector of the form

$$F = b_1 : \text{agg}_1(a_1), \ldots, b_k : \text{agg}_k(a_k)$$

for attributes a_i. Then we define F_g as

$$F_g = b_1 : \text{agg}_1(g.a_1), \ldots, b_k : \text{agg}_k(g.a_k)$$

and introduce the following abbreviation:

$$\Gamma_{G;F}(e) := \Gamma_{G;g;F_g}(e).$$

The notation of the left-hand side will be used for the rest of the paper. Examples for the grouping operator can be found in Appendix E.

The result of Γ is always duplicate-free.

2.5 Groupjoin Operator

The *groupjoin* is defined as follows:

$$\begin{aligned} e_1 \mathbin{⟕}_{A_1\theta A_2;g:f} e_2 \quad := \quad & \{y \circ [g : G] | y \in e_1, \\ & G = f(\{x | x \in e_2, y.A_1\theta x.A_2\}_b)\}_b \end{aligned}$$

Thus, each tuple t_1 in e_1 is extended by a new attribute g whose value is the result of applying a function f to a bag. This bag contains all tuples from e_2 which join on $A_1\theta A_2$ with t_1.

Similar to grouping, we will use $⟕_{q;g;F}$ to abbreviate $\Pi_{\overline{g}}(\chi_F(e_1 ⟕_{q;g:\text{id}} e_2))$, and $⟕_{q;F}$ to abbreviate $⟕_{A;g;F}$. In both cases, F must be an aggregation vector with $\mathcal{F}(F) = \{g\}$. An SQL notation variant of the groupjoin is defined as $e_1 ⟕_{q;F} e_2 := e_1 ⟕_{q;F_g} e_2$, where the requirements for F and F_g are the same as for unary grouping. The notation on the left-hand side will be used for the rest of the paper. Examples for the groupjoin operator can be found in Appendix E.

Since the reader most likely is not familiar with groupjoin, let us give some remarks and pointers on its implementation. Obviously, implementation techniques for the equijoin and the nest operator can be used if θ stands for equality. For the other cases, implementations based on sorting seem promising. One could also consider implementation techniques for non-equi joins, e.g., those developed for the band-width join [10]. An alternative is to use θ-tables, which were developed for efficient aggregate processing [9]. Implementation techniques for groupjoin have also been discussed in [5, 14].

In System T, we implemented the groupjoin for $e_1 ⟕_{a_1=a_2;F} e_2$ as follows. In the first phase, all tuples from e_1 are hashed on a_1 and inserted into a hash table. Thereby, the attributes in $\mathcal{F}(F)$, which contain the results of aggregations, are initialized. For example, for $c_i : \text{sum}(b_i)$, c_i is initialized with 0. In the second phase, all tuples t_2 in e_2 are hashed on a_2 and a lookup into the hashtable is performed. If a tuple t_1 from e_1 which fulfills the join predicate $t_1.a_1 = t_2.b_2$ is found, the attributes in $\mathcal{F}(F)$ are advanced according to the aggregation functions and the values provided by t_2. For example, if $c_i : \text{sum}(b_i)$, c_i is increased by the amount provided by b_i. In the last phase, the entries in the hashtable are pushed into the next operator. Thereby, a final aggregation step may be performed, for example for average.

Note that the groupjoin produces a duplicate-free result if and only if its left input is duplicate-free. More specifically, the left input of a groupjoin can be reconstructed by applying a projection to the result of the groupjoin:

$$\Pi_{\mathcal{A}(e_1)}(e_1 ⟕_{G;G} e_2) \equiv e_1. \tag{1}$$

3. ALGEBRAIC EQUIVALENCES

3.1 Simple Observations about Grouping

As can be seen from the definition of the grouping operator, it is a generalization of duplicate elimination. If we apply a grouping with an empty aggregation vector, then it is equivalent to a duplicate elimination. In other words,

$$\Pi^D_A(e) \equiv \Gamma_{A;()}(e) \tag{2}$$

holds for any set of attributes A with $A \subseteq \mathcal{A}(e)$. As a consequence, we have to ask ourselves whether there exists a property generalizing idempotency that holds for grouping. This is indeed the case. Let F be an aggregation vector decomposable into F^1 and F^2, and G and G^+ be two sets of grouping attributes with $G \subseteq G^+$. Then

$$\Gamma_{G;F}(e) \equiv \Gamma_{G;F^2}(\Gamma_{G^+;F^1}(e)) \tag{3}$$

holds, since we can first group at a finer granularity and then combine finer groups to the groups derived from grouping by G. We can even go a step further in the presence of functional dependencies.

Assume the functional dependency $G \to G'$ holds for two sets of grouping attributes G and G'. Then, the equivalence

$$\Gamma_{G;F}(e) \equiv \Pi_{G \cup \mathcal{A}(F)}(\Gamma_{G \cup G';F}(e)) \tag{4}$$

holds, since the groups and their contents are the same in both cases. This equivalence can also be found under the name *simplify group-by* in a paper by Tsois and Sellis [18]. A slightly more general version for any function f also holds:

$$\Gamma_{G;g:f}(e) \equiv \Pi_{G \cup \{g\}}(\Gamma_{G \cup G';g:f}(e)) \tag{5}$$

Equation 4 can be simplified if, in addition to $G \to G'$, $G \subseteq G'$ holds:

$$\Gamma_{G;F}(e) \equiv \Pi_{G \cup \mathcal{A}(F)}(\Gamma_{G';F}(e)). \tag{6}$$

Assume that the functional dependencies $H \to G$ and $G \to H$ hold. Then, it should not make any difference whether we group by H or G. The only problem we have to solve is that H might not contain all attributes of G (or vice versa). However, since $H \to G$, any attribute $g \in (G \setminus H)$ has only one possible value per group if we group by H. Thus, we can simply copy this value. We do so by adding a new aggregation function cpf(g), which copies the value g of the first tuple seen for a group. This is determistic, since all tuples in a group have the same value for g (as $H \to G$). Thus, to make sure that all values of G are extracted if we group according to H, we extend a given aggregation vector F as follows. Assume $(G \setminus H) = \{g_1, \ldots, g_k\}$. Then, we define $F \circ (G \setminus H)$ as $F \circ (g_1 : \text{cpf}(g_1), \ldots, g_k : \text{cpf}(g_k))$. Using this definition, we can state the equivalence

$$\Gamma_{G;F}(e) \equiv \Pi_C(\Gamma_{H;F \circ (G \setminus H)}(e)), \tag{7}$$

which holds if $H \to G$ and $C = G \cup \mathcal{A}(F)$. This equivalence allows to determine some set H with $H \to G$ such that grouping on H might become cheaper compared to grouping on G, e.g., if the number of grouping attributes is minimized.

Assume that every group consists of only one tuple, i.e., $\Pi^D_G(e) = \Pi_G(e)$ for a set of attributes G with $G \subseteq \mathcal{A}(e)$. If $G \to \text{TID}(e)$, we can replace a grouping by a map:

$$\Gamma_{G;F}(e) \equiv \Pi_C(\chi_{\hat{F}}(e)) \tag{8}$$

with $C = G \cup \mathcal{A}(F)$. Tsois and Sellis call this equivalence *remove-group-by* [18].

3.2 Simple Observations about the Groupjoin

First, let us observe that the groupjoin is redundant. It can be expressed by an outerjoin and a grouping. For $i = 1, 2$, let e_i be expressions, $J_i \subseteq e_i$ be join attributes, and F be an aggregation vector. Define $C = \mathcal{A}(e_1) \cup \mathcal{A}(F)$. Then,

$$e_1 \bowtie_{J_1=J_2;F} e_2 \equiv \Pi_C(e_1 \bowtie^{F(\emptyset)}_{J_1=J_2} \Gamma_{J_2;F}(e_2)) \tag{9}$$

holds if $F(\emptyset) = F(\{\perp_{\mathcal{A}(e_2)}\})$. This holds in SQL-92 for min, max, sum, count(a), but not count(*). More precisely, count(*) yields 0 if the input is the empty set, and 1 if it is applied to some null-tuple. Thus, the right-hand side yields 0 for empty groups, whereas it should produce 1. Obviously, this problem can easily be fixed in the left outerjoin by using the correct default value of 1 for all attributes containing the result of a count(*). The equivalence as such follows directly from the definition of the groupjoin.

Using the above notation, the equivalence

$$\begin{aligned} & \Pi_C(e_1 \bowtie_{J_1=J_2} \Gamma_{J_2;F}(e_2)) \\ \equiv \quad & \sigma_{c_2>0}(e_1 \bowtie_{J_1=J_2;F \circ (c_2:\text{count}(*))} e_2) \end{aligned} \tag{10}$$

is a direct consequence of the above equivalence. Here, the condition $F(\emptyset) = F(\{\perp_{\mathcal{A}(e_2)}\})$ can be omitted, since empty groups are eliminated by the selection $\sigma_{c_2>0}$.

3.3 Groupjoin vs. Group/Left Outerjoin

We now come to the first major equivalence. It will replace a sequence of a left outerjoin and a group by a groupjoin. We first fix the notation. For $i = 1, 2$, let e_i be algebraic expressions and $J_1 = J_2$ be a join predicate, such that for the join attributes $J_i \subseteq \mathcal{A}(e_i)$ holds. For a set of grouping attributes G, define $G_i = G \cap \mathcal{A}(e_i)$ and $G^+_i = G_i \cup J_i$. Further, let F be a splittable and decomposable aggregation vector with $\mathcal{F}(F) \subseteq \mathcal{A}(e_2)$. We denote by C the set of attributes occurring in the result, i.e., $C = G \cup \mathcal{A}(F)$. Then, the equivalence

$$\Gamma_{G;F}(e_1 \bowtie_{J_1=J_2} e_2) \equiv \Pi_C(e_1 \bowtie_{J_1=J_2;F} e_2), \tag{11}$$

holds under the conditions that

1. $G \to G^+_2$ and $G_1, G^+_2 \to \text{TID}(e_1)$ hold in $e_1 \bowtie_{J_1=J_2} e_2$,
2. $J_2 \to G^+_2$ holds in e_2,
3. $\mathcal{F}(F) \subseteq \mathcal{A}(e_2)$, and
4. $F(\emptyset) = F(\{\perp_{\mathcal{A}(e_2)}\})$.

We discuss these conditions to provide the intuition behind them. The two conditions under 1. stem from the main theorem of Yan and Larson in [21]. They assure that a grouping can be pushed into a regular join. In our context, the condition $G_1, G^+_2 \to \text{TID}(e_1)$ assures that no two tuples from e_1 belong to the same group. This is necessary since the groupjoin on the right-hand side provides exactly one output tuple for each input tuple of e_1. The condition $G \to G^+_2$ implies that grouping by G^+_2 is not finer grained than grouping by G, which would lead to problems.

In case the second condition ($J_2 \to G^+_2$) is not fulfilled, we would have more groups on the left-hand side than on the right-hand side of our equivalence, which would violate it. This is easy to see, if we add to G an evil attribute from e_2, which is not functionally determined by J_2.

The importance of the functional dependencies is illustrated in Appendix E.

The third condition ($\mathcal{F}(F) \subseteq \mathcal{A}(e_2)$) can be relaxed as we will see in the appendix (see Equation 14). The fourth condition follows from the discussion of Equation 9.

Equation 11 is important since it allows us to replace a unary grouping and a left outerjoin by a groupjoin. This is very beneficial in several scenarios. Consider just the one where all these operators have a hash-based implementation in a main-memory setting. Then, the left-hand side requires to build two hash tables, whereas the right-hand side requires to build only one. Further, no intermediate result tuples for the outerjoin have to be built.

3.4 Groupjoin vs. Group/Join

We now come to the second major equivalence. It will replace a sequence of a join and a group by a groupjoin. Given the notations of the previous subsection, the equivalence

$$\Gamma_{G;F}(e_1 \bowtie_{J_1=J_2} e_2) \equiv \Pi_C(\sigma_{c_2>0}(e_1 \bowtie\!\!\!\!\Gamma_{J_1=J_2;F \circ (c_2:\mathrm{count}(*))} e_2)), \quad (12)$$

holds under the conditions that

1. $G \rightarrow G_2^+$ and $G_1, G_2^+ \rightarrow \mathrm{TID}(e_1)$ hold in $e_1 \bowtie_{J_1=J_2} e_2$
2. $J_2 \rightarrow G_2^+$ holds in e_2, and
3. $\mathcal{F}(F) \subseteq \mathcal{A}(e_2)$.

The intuition behind these conditions is the same as for the previous equivalence. The importance of the functional dependencies is illustrated in Appendix E. The fourth condition could be omitted, since empty groups are eliminated by the selection $\sigma_{c_2>0}$. Equation 12 is beneficial under similar circumstances as Equation 11.

3.5 The Typical Situation

The situation we find for all TPC-H queries and which we assume is typical, can be described as follows. J_1 is a key of e_1. This is not unlikely, since most joins in SQL are key/foreign-key joins. Further, the set of grouping attributes typically contains the join attributes J_1, i.e., $J_1 \subseteq G$. Last, all grouping attributes are from e_1, i.e., $G \subseteq \mathcal{A}(e_1)$. It is easy to check that in this case the first two conditions of the main equivalences are trivially fulfilled.

4. APPLICATION

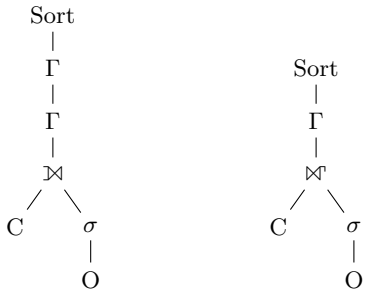
4.1 Groupjoin vs. Group/Outerjoin

Unfortunately, Query 13 is the only one in TPC-H containing an outerjoin. It looks as follows:

```
select    c_count, count(*) as custdist
from      (select   c_custkey, count(o_orderkey) as c_count
           from     customer left outer join
                    orders on c_custkey = o_custkey
           and      o_comment not like
                    '%special%requests%'
           group by c_custkey
          ) as c_orders (c_custkey, c_count)
group by  c_count
order by  custdist desc, c_count desc
```

A straightforward translation into the algebra yields:

$$\begin{aligned} Q_{13} &\equiv \Gamma_{\text{c\_count;custdist:count}(*)}(e_1) \\ e_1 &:= \Gamma_{\text{c\_custkey;c\_count:count}(*)}(e_2) \\ e_2 &:= \text{Customer} \bowtie_{\text{c\_custkey=o\_orderkey}} e_3 \\ e_3 &:= \sigma_{\text{o\_comment not like '\%special\%requests\%'}}(\text{Order}) \end{aligned}$$



Plan without groupjoin Plan with groupjoin

Figure 2: Plans for Query 13.

The plan of this translation is sketched in Figure 2. We left out the details like the grouping attributes, selection and join predicates, since they can easily be derived from the query.

Since c_custkey is the key of the customer relation, we can apply Equation 11 to expression e_2, which yields

$$e_2 \equiv \text{Customer} \bowtie\!\!\!\!\Gamma_{\text{c\_custkey=o\_orderkey;c\_count:count}(*)} e_3$$

A glance at Figure 5 shows that the effort was worthwhile: the query's execution time could be improved by more than a factor of 3. As a general rule, the improvement achieved by applying Equation 11 grows with the ratio of the number of tuples after the left outerjoin compared to the number of tuples produced by the group operator. Since the outerjoin may be much more expensive than the join or the grouping, improvements by a factor larger than 2 are possible. The outerjoin in Query 13 produces 1,533,923 tuples. Since the query needs attributes from both its inputs, 1,533,923 intermediate tuples have to be constructed. These costs are also saved if the groupjoin is used.

4.2 Groupjoin vs. Group/Join: Query 3

There are numerous queries in the TPC-H benchmark which feature a grouping following a join. Some are simple, some are complex. To illustrate the application of Equation 12, we take a look at a simpler (and shorter) one. Query 3 retrieves the first 10 tuples returned by

```
select    l_orderkey, o_orderdate, o_shippriority
          sum(l_extendedprice*(1-l_discount)) as revenue,
from      customer, orders, lineitem
where     c_mktsegment = 'BUILDING'
          and c_custkey = o_custkey
          and l_orderkey = o_orderkey
          and o_orderdate < date '1995-03-15'
          and l_shipdate > date '1995-03-15'
group by  l_orderkey, o_orderdate, o_shippriority
order by  revenue desc, o_orderdate
```

Translation of this query into the algebra yields

$$\begin{aligned} Q_3 &\equiv \mathrm{TopK}_{10}(e_1) \\ e_1 &\equiv \Gamma_{\text{o\_orderkey;revenue:sum}(\cdot)}(e_2) \\ e_2 &\equiv e_3 \bowtie_{\text{l\_orderkey=o\_orderkey}} e_4 \\ e_3 &\equiv \sigma_{\text{l\_shipdate}>\text{1995-03-15}}(\text{Lineitem}) \\ e_4 &\equiv e_5 \ltimes_{\text{o\_custkey=c\_custkey}} e_6 \\ e_5 &\equiv \sigma_{\text{o\_orderdate}<\text{1995-03-15}}(\text{Order}) \\ e_6 &\equiv \sigma_{\text{c\_mktsegment='BUILDING'}}(\text{Customer}) \end{aligned}$$

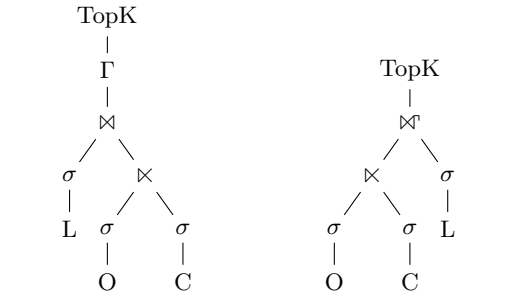


Figure 3: Plans for Query 3.

This plan is also sketched in Figure 3.

We exploited the fact that the query states that l_orderkey = o_orderkey, and that o_orderkey is the key of the order relation. Additionally, we applied Equation 6 to keep the set of grouping attributes small. As a consequence, the sole grouping attribute is o_orderkey, which is the key of the order relation. Thus, the preconditions of Equation 12 are fulfilled and we can rewrite e_1 to

$$e_1 \quad \equiv \quad e_4 \bowtie\Gamma_{\text{o\_orderkey;revenue:sum}(\cdot)} e_3$$

The resulting plan is sketched in Figure 3. From Figure 5, we see that the plan without the groupjoin operator is about 50% slower than the plan with the groupjoin. This is not bad, but also not really impressive. As a general rule, the improvement achieved by applying Equation 12 grows with the ratio of the number of tuples after the join compared to the number of tuples produced by the group operator. To illustrate higher gains, we now briefly sketch a more complex example.

4.3 Groupjoin vs. Group/Join: Query 21

Query 21 retrieves the first 100 tuples returned by

```
select   s_name, count(*) as numwait
from     supplier, lineitem l1, orders, nation
where    s_suppkey = l1.l_suppkey
         and o_orderkey = l1.l_orderkey
         and o_orderstatus = 'F'
         and l1.l_receiptdate > l1.l_commitdate
         and exists (select *
                     from   lineitem l2
                     where  l2.l_orderkey = l1.l_orderkey
                     and    l2.l_suppkey <> l1.l_suppkey)
         and not exists (
                     select *
                     from   lineitem l3
                     where  l3.l_orderkey = l1.l_orderkey
                     and    l3.l_suppkey <> l1.l_suppkey
                     and    l3.l_receiptdate > l3.l_commitdate)
         and s_nationkey = n_nationkey
         and n_name = 'SAUDI ARABIA'
group by s_name
order by numwait desc, s_name
```

This query contains two nested subqueries. We decided to apply the excellent unnesting technique developed by Bellakonda et al.[1]. The resulting plan of this translation is shown in Figure 4. The plan without a groupjoin (left-hand side) has two group operators. The topmost group operator corresponds to the group-by clause in Query 21. It groups by s_name and calculates numwait. The same grouping operator occurs also in the plan with the groupjoin (right-hand side). The main idea of Bellakonda et al. is to coalesce the two subqueries into one. This subquery calculates the number of lineitem tuples which satisfy the conditions of the first subquery, and the number of lineitem tuples satisfying the conditions of the second subquery. This calculation is done in the bottommost group operator of the plan on the left-hand side of Figure 4. These numbers are then used in the predicate contained in the selection following the bottommost grouping. The reader is referred to the original paper for a detailed discussion. For our purposes, it sufficies to know that Equation 12 can be applied to introduce a groupjoin. The performance numbers for both plans are given in Figure 5. As we can see, the performance increases by more than a factor of three. The reason why the plan without a groupjoin is so expensive can be seen from the plan itself. Ignoring details, the plan joins lineitem with lineitem, and lineitem is by far the largest relation in the TPC-H benchmark.

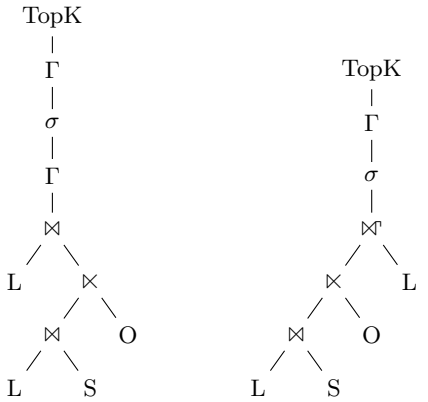


Figure 4: Plans for Query 21.

System T			
X5680@3.33GHz			
TPC-H (SF=1)			
	with ⋈Γ	without ⋈Γ	alternative
Q	t(ms)	t(ms)	t(ms)
13	84	278	
3	70	104	
21	127	500	
5	59	68	
9	212	222	192
10	51	74	
16	45	49	
17	33	34	
20	37	37	
total	1295	1932	all 22 queries of TPC-H

Figure 5: Performance results.

5. EVALUATION

In order to determine the execution times for the different plans, we used our System T, which is a main memory row store. While developing System T, special attention was given to the physical algebra. A carefully crafted push algebra yields sufficient speed. We ran all experiments on a server with an Intel X5680 at a clock rate of 3.33 GHz. The server ran SUSE Linux, and System T was compiled using g++ -O3. The performance results are presented in Figure 5. We used TPC-H with scaling factor 1. All queries ran in single thread mode. Figure 5 gives the execution times for all queries of TPC-H for which one of our equivalences is applicable. The first two rows show the impressive improvements for Query 13, where Equation 11 was applied, and for Query 3, to which Equation 12 was applied. In both cases, a factor of more than three was gained. The remaining queries allow for the application of Equation 12. They do not benefit from the transformation in any impressive way. However, it is important to note that the transformation is always beneficial. Thus, the query optimizer does not have to consider alternatives, which is a very nice feature.

Let us now discuss Query 9. Here, the application of Equation 12 results only in a minor performance improvement. However, there exists another plan with a different join order, to which Equation 12 cannot be applied, which is slightly better than both the other plans. Thus, even if one of the equivalences is applicable to some plan, the resulting plan may not be globally optimal. In a sense, this is what one might have expected anyway, but we like to stress this point. A future challenge will thus be to integrate the new equivalences into a state-of-the-art plan generator like DP_{hyp} [15], which is already capable of handling groupjoins.

The impact of introducing groupjoins is notable on the total query execution time for all TPC-H queries. If no groupjoin is used, the total is 1932 ms, whereas it is 1295 ms if we introduce the groupjoin whenever possible. Thus, this technique saves about 33 % of the total TPC-H execution time.

6. CONCLUSION

We have seen that merging a join followed by a group-by into one groupjoin operator significantly speeds up TPC-H. Since the groupjoin is also useful for decorrelating nested queries and expressing advanced grouping constructs like cube-by and group-by grouping sets, we strongly believe that it should become a standard operator in relational DBMSs. Furthermore, we hope that the groupjoin will find its way into textbooks.

Acknowledgement. We thank Simone Seeger for her help preparing the manuscript. We also wish to thank the anonymous referees for the helpful comments.

7. REFERENCES

[1] S. Bellakonda, R. Ahmed, A. Witkowski, A. Amor, M. Zait, and C.-C. Lin. Enhanced subquery optimization in Oracle. In *VLDB*, pages 1366–1377, 2009.

[2] S. Bitzer. Design and implementation of a query unnesting module in Natix. Master's thesis, U. of Mannheim, 2007.

[3] M. Brantner, N. May, and G. Moerkotte. Unnesting scalar SQL queries in the presence of disjunction. In *ICDE*, pages 46–55, 2007.

[4] R. Cattell, D. Barry, M. Berler, J. Eastman, D. Jordan, C. Russel, O. Schadow, T. Stanienda, and F. Velez, editors. *The Object Database Standard: ODMG 3.0*. Morgan Kaufmann, 1999. Release 3.0.

[5] D. Chatziantoniou, M. Akinde, T. Johnson, and S. Kim. The MD-Join: An Operator for Complex OLAP. In *ICDE*, pages 524–533, 2001.

[6] S. Cluet and G. Moerkotte. Nested queries in object bases. In *Proc. Int. Workshop on Database Programming Languages*, pages 226–242, 1993.

[7] S. Cluet and G. Moerkotte. Classification and optimization of nested queries in object bases. In *BDA*, pages 331–349, 1994.

[8] S. Cluet and G. Moerkotte. Classification and optimization of nested queries in object bases. Technical Report 95-6, RWTH Aachen, 1995.

[9] S. Cluet and G. Moerkotte. Efficient evaluation of aggregates on bulk types. In *Proc. Int. Workshop on Database Programming Languages*, 1995.

[10] D. DeWitt, J. Naughton, and D. Schneider. An evaluation of non-equijoin algorithms. In *VLDB*, pages 443–452, 1991.

[11] A. Klug. Access paths in the "ABE" statistical query facility. In *SIGMOD*, pages 161–173, 1982.

[12] A. Klug. Equivalence of relational algebra and relational calculus query languages having aggregate functions. *Journal of the ACM*, 29(3):699–717, 1982.

[13] N. May, S. Helmer, and G. Moerkotte. Strategies for query unnesting in XML databases. *TODS*, 31(3):968–1013, 2006.

[14] N. May and G. Moerkotte. Main memory implementations for binary grouping. In *Int. XML Database Symp. (XSym)*, pages 162–176, 2005.

[15] G. Moerkotte and T. Neumann. Dynamic programming strikes back. In *SIGMOD*, pages 539–552, 2008.

[16] R. Nakano. Translation with optimization from relational calculus to relational algebra having aggregate funktions. *TODS*, 15(4):518–557, 1990.

[17] H. Steenhagen, P. Apers, and H. Blanken. Optimization of nested queries in a complex object model. In *EDBT*, pages 337–350, 1994.

[18] A. Tsois and T. Sellis. The generalized pre-grouping transformation: Aggregate-query optimization in the presence of dependencies. In *VLDB*, pages 644–655, 2003.

[19] G. von Bültzingsloewen. Optimizing SQL queries for parallel execution. *ACM SIGMOD Record*, 18:17–22, 1989.

[20] G. von Bültzingsloewen. *Optimierung von SQL-Anfragen für parallele Bearbeitung*. PhD thesis, University of Karlsruhe, 1990. in German.

[21] W. Yan and P.-A. Larson. Performing group-by before join. In *ICDE*, pages 89–100, 1994.

[22] W. Yan and P.-A. Larson. Eager aggregation and lazy aggregation. In *VLDB*, pages 345–357, 1995.

APPENDIX

The appendix is organized as follows. We first repeat a result derived by Yan and Larson. Then, we state some simple equivalences which will be needed for the proofs. Since these

are easy to prove, we omit their proof. Finally, we present a lemma and the proofs of our main equivalences.

A. CONVENTIONS

For the remainder of this section, let e_1 and e_2 be two algebraic expressions. Let $q \equiv J_1 = J_2$ be a join predicate with $J_i \subseteq \mathcal{A}(e_i)$ and G be the set of grouping attributes. Define $G_i = G \cap \mathcal{A}(e_i)$, $G_i^+ = G_i \cup J_i$, and $G^+ = G \cup J_1 \cup J_2$. Let F be an aggregation vector. We assume that F is splittable and decomposable. Define $C = \mathcal{A}(F) \cup G$.

B. GROUP AND JOIN

The following equivalence corresponds to the main theorem of Yan and Larson [22]. It states that

$$\begin{aligned} & \Gamma_{G;F}(e_1 \bowtie_q e_2) \\ \equiv\ & \Gamma_{G;(F_1 \otimes c_2) \circ F_2^2}(e_1 \bowtie_q \Gamma_{G_2^+;F_2^1 \circ (c_2:\mathrm{count}(*))}(e_2)) \end{aligned} \quad (13)$$

holds if F is splittable into F_1 and F_2, and F_2 is decomposable into F_2^1 and F_2^2. The proof can be found in [22].

We now wish to eliminate the top-most grouping operator on the right-hand side of Equation 13. We can do so by applying Equation 8, which requires $G^+ \to \mathrm{TID}(e_1)$. If further $G \to G^+$ holds, then there is only one tuple per group in the outermost grouping operator on the right-hand side of Equation 13 and we can apply Equation 6. Thus, under these two conditions we can derive the following equivalence:

$$\begin{aligned} & \Gamma_{G;F}(e_1 \bowtie_q e_2) \\ \equiv^{13}\ & \Gamma_{G;(F_1 \otimes c_2) \circ F_2^2}(e_1 \bowtie_q \Gamma_{G_2^+;F_2^1 \circ (c_2:\mathrm{count}(*))}(e_2)) \\ \equiv^{6}\ & \Pi_C(\Gamma_{G^+;(F_1 \otimes c_2) \circ F_2^2}(e_1 \bowtie_q \Gamma_{G_2^+;F_2^1 \circ (c_2:\mathrm{count}(*))}(e_2))) \\ \equiv^{8}\ & \Pi_C(\chi_{\widehat{(F_1 \otimes c_2) \circ F_2^2}}(\\ & e_1 \bowtie_q \Gamma_{G_2^+;F_2^1 \circ (c_2:\mathrm{count}(*))}(e_2))) \end{aligned} \quad (14)$$

This equivalence corresponds to the main theorem of Yan and Larson in [21]. This can be further simplified if $\mathcal{F}(F) \subseteq \mathcal{A}(e_2)$:

$$\begin{aligned} & \Gamma_{G;F}(e_1 \bowtie_q e_2) \\ \equiv^{14}\ & \Pi_C(\chi_{\widehat{(F_1 \otimes c_2) \circ F_2^2}}(e_1 \bowtie_q \Gamma_{G_2^+;F_2^1 \circ (c_2:\mathrm{count}(*))}(e_2))) \\ \equiv\ & \Pi_C(e_1 \bowtie_q \Gamma_{G_2^+;F}(e_2)) \end{aligned} \quad (15)$$

C. GROUP AND UNION

Let e_1 and e_2 be two expressions with $\mathcal{A}(e_1) = \mathcal{A}(e_2)$. Further, let $G \subseteq \mathcal{A}(e_1)$ be a set of grouping attributes and F an aggregation vector. If $(\Pi_G(e_1) \cap \Pi_G(e_2)) = \emptyset$, then

$$\Gamma_{G;F}(e_1 \cup e_2) \equiv \Gamma_{G;F}(e_1) \cup \Gamma_{G;F}(e_2). \quad (16)$$

If $(\Pi_G(e_1) \cap \Pi_G(e_2)) \neq \emptyset$ and F is decomposable into F^1 and F^2, then

$$\Gamma_{G;F}(e_1 \cup e_2) \equiv \Gamma_{G;F^2}(\Gamma_{G;F^1}(e_1) \cup \Gamma_{G;F^1}(e_2)). \quad (17)$$

This equivalence also holds if $(\Pi_G(e_1) \cap \Pi_G(e_2)) = \emptyset$.

D. MAIN PROOF

Preconditions and Proof Outline

Several equivalences are only valid if certain preconditions hold and equivalences are used to prove other equivalences. For convenience, we thus provide two tables. The first table lists all preconditions of our main equivalences and the equivalences which require them. The second table lists all equivalences together with a list of equivalences used in their proofs. These two tables allow the reader to quickly get an overview of the outline of a proof and the provenance of the preconditions. The first table is

preconditions	equivalences requiring them
$G_1, G_2^+ \to \mathrm{TID}(e_1)$	Equation. 8, 14, 15, 11, 12
$G \to G_2^+$	Equation. 6, 14, 15, 11, 12
$J_2 \to G_2$	Equation. 4, 11, 12
$\mathcal{F}(F) \subseteq \mathcal{A}(e_1)$	Equation. 11, 15, 20

and the second table is:

to prove	equivalences used in proof
Equation 14	Equation. 6, 8, 13
Equation 15	Equation 14
Equation 18	Equation. 13, 17
Equation 19	Equation. 8, 18
Equation 20	Equation 19
Equation 11	Equation. 4, 9, 20

A Lemma We start with a lemma:

$$\begin{aligned} & \Gamma_{G;F}(e_1 \bowtie_{J_1=J_2} e_2) \\ \equiv\ & \Gamma_{G;(F_1 \otimes c_2) \circ F_2^2}(\\ & e_1 \bowtie^{F_2^1(\emptyset), c_2:1}_{J_1=J_2} \Gamma_{G_2^+;F_2^1 \circ (c_2:\mathrm{count}(*))}(e_2)). \end{aligned} \quad (18)$$

Proof. We start with

$$\begin{aligned} & \Gamma_{G;F}(e_1 \bowtie_q e_2) \\ \equiv^{\mathrm{Def}}\ & \Gamma_{G;F}((e_1 \bowtie_q e_2) \cup ((e_1 \triangleright_q e_2) \times E_\perp)) \\ \equiv^{17}\ & \Gamma_{G;F_1^2,F_2^2}(\Gamma_{G;F_1^1,F_2^1}(e_1 \bowtie_q e_2) \\ & \cup \Gamma_{G;F_1^1,F_2^1}((e_1 \triangleright_q e_2) \times E_\perp)), \end{aligned}$$

where $E_\perp = \{\perp_{\mathcal{A}(e_2)}\}$. Applying Equation 13 to the left argument of the union results in

$$\begin{aligned} & \Gamma_{G;F_1^1,F_2^1}(e_1 \bowtie_q e_2) \\ \equiv^{13}\ & \Gamma_{G;(F_1^1 \otimes c_2) \circ F_2^2}(e_1 \bowtie_q \Gamma_{G_2^+;F_2^1 \circ (c_2:\mathrm{count}(*))}(e_2)). \end{aligned}$$

Applying Equation 13 to the right argument of the union yields

$$\begin{aligned} & \Gamma_{G;F_1^1,F_2^1}((e_1 \triangleright_q e_2) \times E_\perp)) \\ \equiv^{13}\ & \Gamma_{G;(F_1^1 \otimes c_2) \circ F_2^2}((e_1 \triangleright_q e_2) \times \Gamma_{G_2^+;F_2^1 \circ (c_2:\mathrm{count}(*))}(E_\perp)) \\ \equiv\ & \Gamma_{G;(F_1^1 \otimes c_2) \circ F_2^2}((e_1 \triangleright_q \Gamma_{G_2^+;F_2^1 \circ (c_2:\mathrm{count}(*))}(e_2)) \\ & \times \Gamma_{G_2^+;F_2^1 \circ (c_2:\mathrm{count}(*))}(E_\perp)) \\ \equiv\ & \Gamma_{G;(F_1^1 \otimes c_2) \circ F_2^2}((e_1 \triangleright_q \Gamma_{G_2^+;F_2^1 \circ (c_2:\mathrm{count}(*))}(e_2)) \\ & \times \Pi_{G_2^+ \cup \mathcal{A}(F) \cup \{c_2\}}(\chi_{F_2^1(\emptyset), c_2:1}(E_\perp))) \end{aligned}$$

and the claim follows. □

A Corollary From Equation 18, we can easily derive the following corollary, by applying Equation 8:

$$\begin{aligned} & \Gamma_{G;F}(e_1 \bowtie_{J_1=J_2} e_2) \\ \equiv\ & \Pi_C(\chi_{\widehat{(F_1 \otimes c_2) \circ F_2^2}}(\\ & e_1 \bowtie^{F_2^1(\emptyset)}_{J_1=J_2} \Gamma_{G_2^+;F_2^1 \circ (c_2:\mathrm{count}(*))}(e_2))) \end{aligned} \quad (19)$$

$l_1 : \Gamma_{a;\text{sum}(d)}(R_1 \bowtie_{a=c} S)$

a	sum(d)
1	17

$l_2 : \Gamma_{a,e;\text{sum}(d)}(R_1 \bowtie_{a=c} S)$

a	e	sum(d)
1	1	8
1	2	9

$l_3 : \Gamma_{a;\text{sum}(d)}(R_2 \bowtie_{a=c} S)$

a	sum(d)
1	34

$l_4 : \Gamma_{a;\text{sum}(d)}(R_3 \bowtie_{b=e} S)$

a	sum(d)
1	17

$r_1 : R_1 \bowtie\!\!\!\!\!\!\Gamma_{a=c;\text{sum}(d)} S$

a	sum(d)
1	17

$r_2 : R_1 \bowtie\!\!\!\!\!\!\Gamma_{a=c;\text{sum}(d)} S$

a	sum(d)
1	17

$r_3 : R_2 \bowtie\!\!\!\!\!\!\Gamma_{a=c;\text{sum}(d)} S$

a	sum(d)
1	17
1	17

$r_4 : R_3 \bowtie\!\!\!\!\!\!\Gamma_{b=e;\text{sum}(d)} S$

a	b	sum(d)
1	1	8
1	2	9

Figure 8: Left- and right-hand sides.

R_1

a
1

R_2

a
1
1

R_3

a	b
1	1
1	2

S

c	d	e
1	8	1
1	9	2

Figure 6: Example relations.

$m_1 : R_1 \bowtie_{a=c} S$

a	c	d	e
1	1	8	1
1	1	9	2

$m_2 : R_2 \bowtie_{a=c} S$

a	c	d	e
1	1	8	1
1	1	9	2
1	1	8	1
1	1	9	2

$m_3 : R_3 \bowtie_{b=e} S$

a	b	c	d	e
1	1	1	8	1
1	2	1	9	2

Figure 7: Join results.

Proof of Equation 19

$$
\begin{aligned}
& \Gamma_{G;F}(e_1 \bowtie_q e_2) \\
\equiv^{18} \quad & \Gamma_{G;(F_1 \otimes c_2) \circ F_2^2}(\\
& \quad e_1 \bowtie_q^{F_2^1(\emptyset), c_2:1} \Gamma_{G_2^+;F_2^1 \circ (c_2:\text{count}(*))}(e_2)) \\
\equiv^{8} \quad & \Pi_C(\chi_{\widehat{(F_1 \otimes c_2)} \circ F_2^2}(\\
& \quad e_1 \bowtie_q^{F_2^1(\emptyset)} \Gamma_{G_2^+;F_2^1 \circ (c_2:\text{count}(*))}(e_2)))
\end{aligned}
$$

□

If $\mathcal{A}(F) \subseteq e_2$, Equation 19 can be simplified to

$$
\begin{aligned}
& \Gamma_{G;F}(e_1 \bowtie_{J_1=J_2} e_2) \\
\equiv \quad & \Pi_C(e_1 \bowtie_{J_1=J_2}^{F(\emptyset)} \Gamma_{G_2^+;F}(e_2)) \qquad (20)
\end{aligned}
$$

Proof of Equation 11 We now give the proof of Equation 11. We start with the right-hand side and transform it until we get the left-hand side:

$$
\begin{aligned}
& \Pi_C(e_1 \bowtie\!\!\!\!\!\!\Gamma_{J_1=J_2;F} e_2) \\
\equiv^{9} \quad & \Pi_C(e_1 \bowtie_{J_1=J_2}^{F(\emptyset)} \Gamma_{J_2;F}(e_2)) \\
\equiv^{4} \quad & \Pi_C(e_1 \bowtie_{J_1=J_2}^{F(\emptyset)} \Gamma_{G_2^+;F}(e_2)) \\
\equiv^{20} \quad & \Gamma_{G;F}(e_1 \bowtie_{J_1=J_2} e_2))
\end{aligned}
$$

□

Proof of Equation 12

Equation 12 follows directly from Equation 11. An alternative is to modify the above proof by using Equation 10 instead of Equation 9 and Equation 14 instead of Equation 20.

E. EXAMPLES

Figure 6 contains some relations. The results of some outerjoins ($R_i \bowtie_q S$) with two different join predicates are given in Figure 7. Since all tuples in some R_i always find a join partner, the results of the outerjoins are the same as the corresponding join results. We are now interested in the functional dependencies occurring in the conditions of our main equivalences. Therefore, we discuss four example instances of Equation 12, where at most one of the functional dependencies is violated:

	$G \to G_2^+$	$G_1, G_2^+ \to \text{TID}(e_1)$	$J_2 \to G_2^+$
1	+	+	+
2	+	+	-
3	+	-	+
4	-	+	+

The according instances of the left-hand and right-hand side of Equation 12 are:

	LHS	RHS
1	$\Gamma_{a;\text{sum}(d)}(R_1 \bowtie_{a=c} S)$	$R_1 \bowtie\!\!\!\!\!\!\Gamma_{a=c;\text{sum}(d)} S$
2	$\Gamma_{a,e;\text{sum}(d)}(R_1 \bowtie_{a=c} S)$	$R_1 \bowtie\!\!\!\!\!\!\Gamma_{a=c;\text{sum}(d)} S$
3	$\Gamma_{a;\text{sum}(d)}(R_2 \bowtie_{a=c} S)$	$R_2 \bowtie\!\!\!\!\!\!\Gamma_{a=c;\text{sum}(d)} S$
3	$\Gamma_{a;\text{sum}(d)}(R_3 \bowtie_{b=e} S)$	$R_3 \bowtie\!\!\!\!\!\!\Gamma_{b=e;\text{sum}(d)} S$

The functional dependencies have to be checked on the join results given in Figure 7. In order to help the reader to check the functional dependencies, we provide the following table holding the main attribute sets occurring in our main equivalences:

	G	G_1	G_2	J_2	G_2^+
1	$\{a\}$	$\{a\}$	$\emptyset$	$\{c\}$	$\{c\}$
2	$\{a, e\}$	$\{a\}$	$\{e\}$	$\{c\}$	$\{c, e\}$
3	$\{a\}$	$\{a\}$	$\emptyset$	$\{c\}$	$\{c\}$
4	$\{a\}$	$\{a\}$	$\emptyset$	$\{e\}$	$\{e\}$

Taking a look at Figure 8, we see that both sides of the equivalence give the same result only if none of the functional dependencies is violated.