Joins vs. Links or Relational Join Considered Harmful

Alexandr Savinov
Bosch Software Innovations GmbH, Stuttgarterstr. 130, 71332 Waiblingen, Germany

Keywords: Data Processing, Data Modeling, Join Operation, Links, References, Connectivity.

Abstract: Since the introduction of the relational model of data, the join operation is part of almost all query languages and data processing engines. Nowadays, it is not only a formal operation but rather a dominating pattern of thought for the concept of data connectivity. In this paper, we critically analyze properties of this operation, its role and uses by demonstrating some of its fundamental drawbacks in the context of data processing. We also analyze an alternative approach which is based on the concept of link by showing how it can solve these problems. Based on this analysis, we argue that link-based mechanisms should be preferred to joins as a main operation in data model and data processing systems.

1 INTRODUCTION

A *data model* is a definition of data, that is, it answers the question what is data. It defines how data is organized and how it is manipulated by providing structural elements and operations. For example, the relational model (Codd, 1970) organizes data by using tuples, domains and relations while the functional data model (Sibley and Kerschberg; 1977) does the same by using functions. One of the major concerns in any data model is describing how elements are *related* or *connected* as well as providing means for retrieving related elements. The mechanism of connectivity determines such important aspects as semantic clarity, conciseness of queries, maintainability and performance.

Probably the most wide spread approach to retrieving related elements in a database is based on the operation of relational algebra called *join*. A join takes two or more relations as input and produces a new relation as output. The output relation contains tuples composed of *related* tuples from the inputs. The way they are related is specified in the join condition which is a parameter of the operation.

Although the join operation dominates in the area of data management, there are also alternative approaches. One of them defines how data elements are related and accessed by using the notion of *link* or *reference* (we will use these terms interchangeably because their differences are not important for this paper). A link is a value which identifies a data element (referent) and is used to access it. It is a very simple and natural concept which dominates in programming but is also used in many data processing systems and models.

This paper is devoted to comparing joins and links as alternative mechanisms for accessing related data elements which are based on completely different patterns of thought. We critically analyze join operation and demonstrate that it has some significant drawbacks while links on the other hand can solve these problems. We argue that joins should not be used in data modeling and data processing systems or at least their role should be significantly diminished. We also want to show that links can be used as a basis for a new mechanism that can replace joins and do what joins have been intended for.

The paper has the following layout. Section 2 and Section 3 are devoted to describing joins and links, respectively. Each of these sections starts from describing the corresponding mechanism and then we analyze their more specific properties. Section 4 makes concluding remarks.

2 "WHO IS TO BLAME?" JOINS

2.1 What is in a Join? Common Value

Let us assume that there are two CSV files with a list of employees and departments which might have the followings structure and data:

Savinov, A.
Joins vs. Links or Relational Join Considered Harmful.
DOI: 10.5220/0005932403620368
In *Proceedings of the International Conference on Internet of Things and Big Data (IoTBD 2016)*, pages 362-368
ISBN: 978-989-758-183-0

```
emp, name, dept_id
25, Smith, RD
```

```
dept, mngr, location
RD, 30, Stuttgart
```

These two lines are stored in different files and formally are not connected because there is no indication of any relation between them. However, we can easily see that both of these lines store the same value 'RD' and we also know that this value (semantically) represents the same entity.

This fact of being characterized by the same value is a basis for establishing relationships between data elements. In our example, the fact that one employee and one department are *both* characterized by the value 'RD' is not a coincidence but rather a way to represent that they are connected (Fig. 1). In other words, two data elements are supposed to be related if they have something in common or, more specifically, share the same value. Therefore, this general connectivity principle could be called a common value or *shared value* approach. Importantly, in addition to the two data items being connected, there is a third item stored in both of them and treated as a means of connectivity. There is no way to connect two elements without specifying a third element they share.

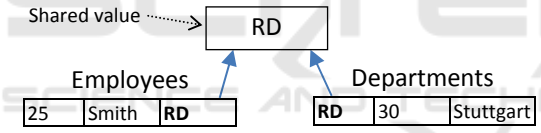

Figure 1: Two elements are related if they share a third element (data value).

Having something in common is a conceptual definition of related elements. It allows us to determine whether two elements are related or not. However, the problem is not only to define related elements but also to *retrieve* them by materializing this conceptual relationship. In the relational algebra, such a mechanism is provided by the *join operation* which is applied to relations and returns a new relation. In addition to input relations, this operation needs a parameter which provides a criterion of connectivity. The output relation will contains only tuples satisfying this condition.

Although joins can specify any condition the combinations of input tuples have to satisfy (theta-join), the most common case is to select only input tuples which contain equal values of some attributes (equijoin). Fig. 2 provides an example of joining two tables `Employees` and `Departments` by producing a new table `Emps_Depts`. Note that every tuple of the result table is a combination of matching tuples from the two input tables which share the same value.

Employees

emp	name	dept_id
25	Alex	RD
26	John	HR
27	Anna	HR
emp	name	dept_id
25	Alex	**RD**

Departments

dept	mngr	location
RD	30	Dresden
HR	20	Berlin
dept	mngr	location
RD	30	Dresden

Emps_Depts

emp	name	dept_id	dept	mngr	locatioin
25	Alex	**RD**	**RD**	30	Dresden
26	John	**HR**	**HR**	20	Berlin
27	Anna	**HR**	**HR**	20	Berlin

Figure 2: Output relation contains combinations of related tuples from input relations.

The idea of using common values for matching data elements has its formal roots in predicate calculus and is used various technologies, for example, deductive databases (Ullman & Zaniolo; 1990) or query languages. If two predicates in a logical expression have the same free variables then they have to be bound to the same value in order for the resulting proposition to be true. For example, given two predicates

```
Employees(emp, ename, city)
Departments(dept, dname, city)
```

we can define a logical expression `Employees(emp, ename, city) & Departments(dept, dname, city)` which will be true only if the `city` variable takes the same value. In this way, we can retrieve all employees who are located in (share) the same city as their department.

2.2 Join is Symmetric

The underlying semantics of the join operation is two elements are defined to be related if they have the same property. This property makes it a *symmetric* operation where all inputs have the same roles. For example, if we look at this join condition

```
Employees.dept_id = Departments.dept
```

then it is not possible to assign a special role to one of the input tables `Employees` or `Departments` (Fig. 3).

Here we see one major problem of joins: in fact, these two tables have different semantic roles and the relationship between them is not symmetric. One indication of asymmetricity is that it is a many-to-one relationship where many employees belong to one department. Another observation is that this

same semantics can be (correctly) represented as an employee record referencing one department (but not vice versa). Also, the shared value is a department identifier and employee identifier. If we change the direction of this relationship then we change the meaning of the connection. Yet, joins are not able to represent this semantics because all input relations have equal rights in a join.

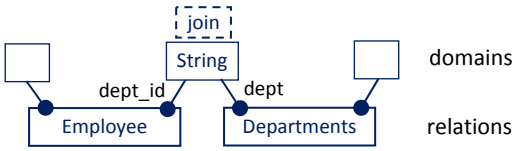

Figure 3: Join is symmetric.

In the relational model, a unit of connectivity (between domains) is one relation and *composition* means joining relations. In other words, join allows to compose or chain relations. Now assume that we have 10 relations and want to retrieve tuples from one of them which are related to tuples in another relation. Formally, we need to build a Cartesian product of these relations by adding also all relevant join conditions. This approach is highly unnatural and very difficult to use in practice (therefore, its use is quite rare). Why we have to include all input relations if we want to retrieve records from only one of them? It is also not obvious what join conditions to use (especially if we do not have FKs declared). It is therefore very easy to produce formally correct but semantically wrong results. And the reason is that the constraints are propagated along a sequence of relations connected by common values. It is probably one of the reasons why the development of the conception of automatic reasoning in the unified relation model (URM) failed (Maier et al., 1984).

It should be noted that there exist a mechanism of foreign keys (FK) that can solve this problem of symmetricity of joins. Indeed, a FK declares one input relation and one output relation which have different roles. If two relations are used in a join then this FK declaration can be used as a semantic specification of our intention in the join operation. Yet, the use of FKs for this purpose has the following drawbacks. First, it is not an original purpose of FKs to describe semantics of joins (FK is a mechanism of imposing constraints). Second, the need in an additional mechanism like FK emphasizes that joins have some limitations. Third, FKs are not enforced by existing models in general and they must not be used in the context of joins in particular. Join is an operation which is used at query time while FK is a declaration which is used at design time. Fourth, it can be difficult to understand how to use FKs in the case of arbitrarily complex join conditions. Essentially, FK is an attempt to introduce a mechanism of links but they have incompatible semantics and therefore their simultaneous use is quite controversial and eclectic. The use of FKs in combination with joins is analogous to introducing constructs for structural programming along with goto operator. Their simultaneous use will result in strange mixtures of different patterns.

2.3 Join is a Cross-cutting Concern

Let us assume that we want to get a list of employees working at some department. It can be done by means of the following query:

```
SELECT E.emp, D.dept, D.location
FROM Employees E, Departments D
WHERE E.dept_id = D.dept
```

An important observation here is that many similar queries will include the same join condition. In other words, this same join condition `E.dept_id=D.dept` will appear in quite many queries which involve these two tables. It is because join conditions describe the details of *how* entities are connected in the model rather than the logic of what needs to be retrieved.

Such fragments of the source code which scatter throughout the whole program or query are referred to as a *cross-cutting concern* (Kiczales et al., 1997). The existence of such repeated fragments of code is an indication of either bad design or impossibility to modularize their logic due to limitations of the language. The main negative consequence is that the same fragment can appear in quite different and unrelated contexts semantically belonging to different levels of organization. As a result, the program or query can become error-prone and difficult to maintain. The solution of this problem is to provide a mechanism for *modularizing* such repeated fragments in a separate modeling construct.

Ideally, a mechanism of connectivity should declare *how* different entities in the whole model are connected independent of where these connections will be used. Yet, in join-based queries, both the logic of the query and the logic of the connections between relations are described together in the same construct. It is a typical example of *mixing* different concerns. On one hand, the main purpose of the query is to *retrieve* employees with the related department information. It is application-specific logic and we do not care how these relations are

connected. On the other hand, this same query involves a fragment which has nothing to do with this application because it describes how the connection between employees and departments is implemented (independent of any queries). These two concerns should be separated but the approach based on joins does not support this separation.

This join-based approach to querying data leads to exposing low-level structure of connections at the higher level of application-specific logic. For example, if later on we will change the way relations (departments and employees) are connected then *all* queries that use it will have to be updated. It is also error-prone because query writers are not necessarily experts in connectivity – they have to know only that these two tables have a specific connection while its implementation should be effectively hidden in a separate module or mechanism. This task of combining connectivity criteria with the business logic of the query is especially difficult in the case of complex multi-table queries. It is also a potential security flaw because a low-level definition of connectivity can be influenced from the higher level of user-oriented operations. This explicit use of join conditions can also result in lower performance because the database engine cannot use optimizations for pre-defined connections but rather has to assume the possibility of any join conditions in any new query.

2.4 Joins do Not Support Types

Let us assume that department identifiers are unique strings and we want to retrieve tuples with the same department:

```
SELECT *
FROM Employees, Departments
WHERE dept_id = dept
```

If we now modify this query by changing its join conditions as follows

```
WHERE dept_id = dept_name
```

then we get a formally valid query which however is semantically wrong because it does not make sense to compare department identifiers with department names. The problem is that `dept_id` is declared as a string but semantically it is a `Department`. Yet, we are not able to declare the correct type of `dept_id`: neither as an attribute type at the level of the model (relations cannot be used as types) nor at the query level in the join condition. Therefore, it is not possible to prevent the user from writing semantically wrong conditions even if we know that they are wrong.

It should be noted that foreign keys could help in this situation. But it is an auxiliary mechanism which exists independently of types (and joins). Both FKs and types are used to constrain sets but do it differently. In our example, if we want to (correctly) declare the `dept_id` attribute as referencing a department then we either add a foreign key that constrain the values by those existing in the `Departments` table or directly use `Departments` as a type. It is obviously duplication of functionality in two mechanisms with significant negative consequences.

3 "WHAT IS TO BE DONE?" LINKS

3.1 What is in a Link? Inclusion

Links or references are used if it is necessary to *represent* some target element within this element. This representation can be then used to *access* the target element. Links are widely used in programming languages to represent and access objects using some kind of a unique value. This value, called reference, can be then stored in other objects. As a result, a program is represented as a graph where nodes are objects and edges are references. Link is also a basic mechanism of the world wide web where text documents can reference other text documents by using their unique identifiers. Users can traverse this graph of documents by following links and retrieving new documents.

Link as a mechanism of connectivity possess the following general properties:

- Link is a *value* (passed by-value only) which uniquely represents some object (referent) and is used to access it. Primary key or similar annotation is a design pattern that can be used to model references but is different from true references because references do not belong to the represented object properties.
- Link is a mechanism of indirectly *including* one object in another object. Foreign keys can be used to declare (annotate) such an inclusion. Yet, it is a design pattern and not a true link declaration. In particular, foreign keys cannot be used for access and do not have types.
- The mechanism of links assumes the existence of three roles: represented object (referent), reference (link), and a referencing object which stores the reference.

- The mechanism of access provided by a reference is hidden in its implementation and is not exposed at the application level where links are used.
- Link is a directed relation. In particular, this means that one (referencing) entity knows about the other (referenced) entity but not vice versa.
- Links on sets can be formally described by using mathematical functions (mappings).

3.2 Link is a Directed Relationship

Links are not supported in the relational model of data. Yet, conceptually we can think of table records as being linked by using the following interpretation. If a record stores one or more values which uniquely identify and can be used to access another record then these values are thought of as a link. For example, if one employee record stores a unique identifier of a department then this means that this employee references this department (Fig. 4). Note however that although the (physical) representation via tables is the same for both joins and links, they have different meanings.

One of the main distinguishing features of links is that the ordering of the roles differs from that used in the join-based model. Links also use three roles: identifier (reference), referenced entity (referent) and referencing entity. However, rather than sharing the same value (department identifier in Fig. 4), the `Employees` table includes an identifier from the `Department` table while which in turn includes a string as identifiers. It is not only a visual ordering – it has significant semantic consequences.

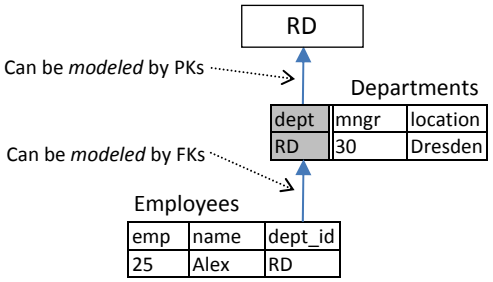

Figure 4: Two tuples are connected by storing in one of them a data value representing the other one.

In contrast to joins, links represent binary directed relationships among data elements with opposite roles for the two arguments. One advantage of such an approach is that links are much easier to compose. For example, given an employee element, we can find the corresponding department address by composing two references:

```
Address addr = emp.department.address;
```

This composition operation (syntactically encoded as dot notation) follows a path in a directed graph of links which is a very natural and semantically unambiguous operation. For comparison, the same task can be solved by applying two join operations:

```
SELECT * FROM
Employees E, Departments D, Addresses A
WHERE E.dept_id=D.dept && D.addr_id=A.addr
```

Here it is difficult to determine the real purpose of this query and the role of its arguments (relations) because the result of the operation is one relation combining tuples from the three input relations. It is also not obvious how to inverse this query.

The difference between link composition and join composition is analogous to that between function composition in mathematics and bindings in predicate calculus. In fact, these are two quite different ways to think about connectivity. The join-based approach assumes that two elements are connected if they both include a third element. The link-based approach assumes that two elements are connected if one of them includes (a reference to) the other.

3.3 Links Modularize and Hide the Access Mechanism

If we want to access related records using the mechanism of joins then we have to provide a specification of *how* the records are related. Moreover, this specification has to be provided for each query in the same statement with other application specific parameters.

Links are much easier to use. In order to retrieve related records it is necessary to provide only one attribute name. For example, given a reference to an employee object, we can get a related department object by simply specifying an attribute name:

```
Departmet dept = emp.department;
```

What is important here is that we actually do not know how departments are retrieved and what is the definition of the relation between departments and employees. This definition and the access mechanism are hidden in the `department` attribute. Effectively, two concerns are principally separated:

- how links are used, and
- how links are defined

This approach provides significant advantages especially for complex systems. We can easily

change the definition of a link without the need to update all its usages. One possible implementation of user-defined links is implemented in DataCommandr (Savinov; 2016). For example, assume that we want to re-design our data model so that an employee can be assigned to more than one department. In this case, the department attribute will have to be redefined so that it returns an element marked as a main department. If we were using joins then this would require modification of all queries that access departments using an employee. In the case of links, it is enough to change only one link definition. References can also be applied to sets in the form of project operation and arrow notation as proposed in the concept-oriented model of data (Savinov; 2014).

3.4 Links Support Types

Type is a set of elements and specifying a type means imposing constraints on possible elements. Any link definition involves two such types: a set of input elements and a set of output elements. Thus links very naturally support typing as an integral part of this mechanism.

For example, if we want to define a link from employees to departments then this means that it has to work for only input elements from the `Employees` set and output elements from the `Departments` set. These constraints can be declared as a function signature:

```
Departments departments(Employees this);
```

They also can be declared as a field type of a class:

```
class Employees {
  Departments departments;
}
```

Note that links allow us to impose type constraints independent of whether the output is a relational domain or another relation. Also, typed links make foreign keys unnecessary.

4 CONCLUSIONS

The main result of this paper is that although join is a powerful formal operation it has the following major drawbacks in the context of data processing:

- Join relies on the semantics of *common values* for representing connectivity which is not very natural semantically and can be counterintuitive when using composition.
- Join is a *cross-cutting concern* because the same join condition can spread over many queries by mixing two concerns: application specific logic of the query and how connectivity is implemented.
- Join does not inherently support *typing* because two related (joined) sets are separated by a third set with common values.

Therefore, the use of joins normally requires high expertise and can lead to error prone and difficult to maintain queries and data models.

We also analyzed an alternative mechanism of links which provides the following benefits:

- Links rely on the semantics of *inclusion* of an identifier of one element into another element which is very natural for describing how things are connected and allows for easily composing and inverting the relationships.
- Links effectively *modularize* and hide the underlying mechanism of access so that the main logic of the query does not involve the description of how the connectivity is implemented. Primary keys can be viewed as one possible design pattern for implementing true references.
- Links are typed by specifying sets for their input and output elements which is precisely what is normally needed. Foreign keys can be viewed as one possible design pattern for implementing true typing.

Taking into account these properties we argue that links should be preferred to joins as a primary mechanism for accessing related elements in databases. Yet, classical references miss some properties which are of crucial importance for data modeling. How links can be revisited in order to overcome these drawbacks will be our focus for future research.

REFERENCES

Codd, E., 1970. A Relational Model for Large Shared Data Banks. *Communications of the ACM*, 13(6), 377-387.

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., Irwin, J., 1997. Aspect-Oriented Programming. *ECOOP'97*, 220-242.

Maier, D., Ullman, J.D., Vardi, M.Y., 1984. On the foundation of the universal relation model. *TODS'84*, 9(2), 283–308.

Savinov, A., 2014. Concept-oriented model. In J. Wang (Ed.), *Encyclopedia of Business Analytics and*

Optimization. IGI Global, 502-511.
Savinov, A., 2016. DataCommandr: Column-Oriented Data Integration, Transformation and Analysis. *Internet of Things and Big Data (IoTBD'2016)*.
Sibley, E.H. & Kerschberg, L., 1977. Data architecture and data model considerations. In *Proceedings of the AFIPS Joint Computer Conferences*. 85-96.
Ullman, J.D., Zaniolo, C., 1990. Deductive databases: achievements and future directions. *ACM SIGMOD Record*, 19(4), 75-82.