Image database management systems design considerations algorithms and architecture
Nes, N.J.

Citation for published version (APA):
Nes, N. J. (2000). Image database management systems design considerations algorithms and architecture

General rights
It is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), other than for strictly personal, individual use, unless the work is under an open content license (like Creative Commons).

Disclaimer/Complaints regulations
If you believe that digital publication of certain material infringes any of your rights or (privacy) interests, please let the Library know, stating your reasons. In case of a legitimate complaint, the Library will make the material inaccessible and/or remove it from the website. Please Ask the Library: http://uba.uva.nl/en/contact, or a letter to: Library of the University of Amsterdam, Secretariat, Singel 425, 1012 WP Amsterdam, The Netherlands. You will be contacted as soon as possible.
Chapter 6

Fitness Join

6.1 Introduction

Join algorithms have a long tradition of interest in the database community. By the late-70s the key algorithms were published [7]. Nested-loop, sort-merge, and hash-based join algorithms have successively been explored extensively to reduce their running time, including their parallel versions. For an overview of these results see [73, 67]. In the object-oriented context joins based on path traversals have been supported using join indices [111] and pointer-based algorithms [33]. The results have been generalized into a methodology for index structures[38, 23].

The common denominator of the join studies is that they largely deal with equi-join conditions. Algorithms with more complex conditions, i.e. theta-joins, have been barely scratched upon [34], let alone their embedding in real-life applications.

Band-joins have been identified as a crucial asset for engineering applications, which require constraints on intervals around the join attribute. A traditional sort-merge algorithm has been put forward as the best attack, which brings the problem back into the realm of the now classical algorithms [34].

More recently, Helmer and Moerkotte [97] have investigated the extension of main-memory join algorithms to deal with subset-join predicates. From an application perspective, near match solutions are considered of more importance than total match. Further progress has been reported for optimizing cross products in [97] and universally quantified predicates in [24].

These join activities are exemplary to better support novel applications in a bottom-up fashion. Namely, the core relational algebra is extended gradually with new features. However, starting from the way novel (non-database) applications are being built, we have encountered the need for fitness joins, which take the form of identifying join pairs optimally placed
or ranked under a given metric condition. The naive solution for a fitness join starts with producing the cross product followed by calculation of a fitness value for each pair using a (sub-)query. Thereafter the best (worst) from the perspective of one of its operands is (are) retained (for ranking).

The contributions of this chapter rest on introducing a reference example to study the class of fitness joins, followed by chartering a road to develop new algorithms and optimization schemes. The search space for effective solutions is trimmed by presenting a concrete operator for fitness joins, called the bounded theta join operator implemented in Monet [9]. The scope for further optimization is illustrated using the mathematical properties as guidance for search heuristics. The bottom line being to extend a query optimizer with a limited form of mathematical reasoning to derive effective processing heuristics.

The remainder of this chapter is organized as follows. Section 6.2 provides a motivational example to study the fitness joins. Section 6.3 illustrates how fitness joins can be handled in both a traditional and extensible database setting. It also introduces the bounded-theta solution in Monet. Section 6.4 charts the contours of mathematical query optimization. Section 6.5 summarizes a concrete implementation and its evaluation. We conclude with challenges for subsequent research.

6.2 Motivation

In this section we introduce the class of joins considered with an artificial example, the ballroom, followed by an indication of the application domains in which they appear.

6.2.1 The Ballroom Example

Consider the database shown in Figure 6.1, which lists the participants of a yearly ballroom contest. All persons are qualified dancers as illustrated by their repertoire. In the training session for the contest it makes sense to identify (or rank) potentially good teaching partners, i.e. those with a good overlap of dance repertoire and partners of similar age. During a dance session it is mandatory to quickly find a partner with the proper dance repertoire on the dance floor when the music changes, because dancing with an inexperienced person does not contribute to the training experience, especially with the Tango. Positions of the dancers are also shown ([x,y]).

These questions can be rephrased to database queries as follows:

- $Q_1$: Find a partner of opposite gender and closest in age.
- $Q_2$: Find a partner with the best overlap in dance repertoire.
- $Q_3$: Find a male partner with a larger dance repertoire.
- $Q_4$: Find the closest Tango partner on the floor.
- $Q_5$: Rank the dance partners by experience.
Let's have a more detailed look at the requirements. $Q_1$ can be answered using the age difference over persons, i.e. $f_1(r, s) = \text{abs}(r.age - s.age)$, and to retain those that minimize this function for all possible couples. Query $Q_2$ calls for comparison of set-based attributes, where we have to map dance capabilities into a relevant information quantifier. Good partners are those that have an identical dance repertoire. The best dance partner would be one that maximizes the fitness function that compares the repertoire sets of couples:

$$f_2(r, s) = \frac{|r.repertoire \cap s.repertoire|}{|r.repertoire \cup s.repertoire|}$$

Choosing a good teacher calls for inspection of the repertoire of the potential partners. Their dance repertoire should exceed your own and the best teacher is selected by repertoire count. This can be answered with the following simple fitness function:

$$f_3(r, s) = \begin{cases} |\text{repertoire}(s.name)| & \text{if } r.repertoire \supset s.repertoire \\ 0 & \text{otherwise} \end{cases}$$

The teacher selection procedure can be further refined by keeping a tally on the experience of each dance performed by a person. However, then we run into a semantic problem as to precisely define by what is meant with a good teacher. The predominant approach is to look at the experience histogram distributions for two repertoire sets and determine a metric similarity. The best teacher is then someone with larger repertoire and a maximal distance in the experience space. This, however, quickly breaks down. An Euclidean metric would favor a super expert on a single dance over someone who has adequate dance experience in multiple dances. The way out of this dilemma is to consider alternative (standard) metrics or
to call upon the user to give a fuzzy mathematical definition of the metric intended.

Query $Q_4$ combines a simple predicate with a spatial term using the Euclidean distance in the ballroom and we retain those minimizing the fitness function.

$$dist(r, s) = \sqrt{(r.x - s.x)^2 + (r.y - s.y)^2}$$

$$f_4(r, s) = \begin{cases} 
\text{dist}(r, s) & \text{if } \text{"tango"} \in r.\text{repertoire} \cap s.\text{repertoire} \\
\text{nil} & \text{otherwise} \\
\end{cases}$$

Ranking the dance partners by experience ($Q_5$) calls upon sorting the possible couples using e.g. the function $f_5(r, s) = s.\text{experience}$ or $f_r(r, s) = s.\text{experience}/|r.\text{repertoire} \cup s.\text{repertoire}|$. Sorting boils down to an iteration over the candidates, finding the next best partner not ranked yet. Although sorting can be implemented with repeated joins, it is certainly not the best solution.

In practice, arbitration may also be required to assure a maximum number of couples on the dance floor, i.e. minimizing the total dissatisfaction. This aspect is not dealt with in this thesis.

### 6.2.2 Fitness Joins

The ballroom examples leads to the class of fitness joins. The basis is formed by two sets of objects R and S organized into a bi-partite graph. This graph is obtained by taking the cross-product over R and S retaining pairs that satisfy a priori selection criteria. Each edge carries a value obtained from evaluation of a fitness function. From this enriched graph we retain all edges that minimize (maximize) the edge weight from the perspective of the first operand.

For example, the figures below represents the bi-partite graphs and fitness join result for the dancers under fitness functions for $f_1$, $f_2$ and $f_4$, respectively. It illustrates that persons may be assigned several candidates and that (in principle) the function is not symmetric.

The functions are built from the standard repertoire of mathematical functions in a DBMS. Furthermore, we have encountered the need to also describe discontinuous and constrained functions, e.g.

$$f(r, s) = \begin{cases} 
Expr_1 & \text{if } Cond_1 \\
Expr_{n-1} & \text{if } Cond_{n-1} \\
Expr_n & \text{otherwise} \\
\end{cases}$$

The function lines being interpreted in sequential order until a condition holds. For example, in case the ballroom lacks experienced dancers of opposite gender, we may relax the partner choice. This fuzzy function could be described as:
6.2. MOTIVATION

![Diagram of relationships between individuals: Brian, Tina, Edward, Mary, Owen, Susan, Peter, and Ruth.](image)

Figure 6.2: Query Fitness Graphs

\[
f(x, y) = \begin{cases} 
1 & \text{if } x.\text{gender} \neq y.\text{gender} \\
& \land x.\text{repertoire} \cap y.\text{repertoire} \neq \emptyset \\
0.7 & \text{if } x.\text{repertoire} \cap y.\text{repertoire} \neq \emptyset \\
0.1 & \text{otherwise}
\end{cases}
\]

Care should be taken on the definition of \( f \). Besides being type correct we ignore all candidate pairs where either \( f(r, s) \) or \( f(r, t) \) is undecided or evaluates to nil. In general, this function can not be mimicked with an SQL CASE statement and the user is forced to create a query batch to simulate the behavior intended.

In the context of an extended relational model the fitness join can now be defined as follows:

**Definition** Let \( R \) and \( S \) be two (object) relations. Then the *minimal* fitness join \( \Join_f \) over \( R \) and \( S \) under function \( f \) is defined by:

\[
R \Join_f S = \{(r,s) | r \in R, r \neq s \in S \land \forall t \neq r \in S \ f(r,s) \leq f(r,t)\}
\]

This definition combines tuples that minimize the fitness function \( f \), it identifies (local) minima in the function space from \( R \). If both arguments to a fitness join are one and the same table then the definition assures that we do not retain the identity pairs as being most fit. Likewise we can pair objects to maximize the fitness function.
**Definition** Let \( R \) and \( S \) be two (object) relations. Then the *maximal* fitness join \( <^f \) over \( R \) and \( S \) under function \( f \) is defined as follows:

\[
R <^f S = \{(r, s) | r \in R, r \neq s \in S \forall t \neq r \in S f(r, s) \geq f(r, t)\}
\]

The consequence of our liberal definition of fitness functions is that, in general, the fitness joins are not symmetric (\( R <^f S \neq S <^f R \)), nor transitive (\( (R <^f S) <^f T \neq R <^f (S <^f T) \)). For a given point there may exist several minima (maxima).

### 6.2.3 Application domains

The Ballroom problem has notably many incarnations in literature. We have also encountered them in several applications developed for Monet [13, 79, 12]. They often appeared as heuristic functions written in a traditional language (c,C++). A pattern emerged that called for better support to analyze object pairs using a concise mathematical formulae, i.e. the *fitness* function, as a necessary step in an iterative process to combine elements into larger semantic units (objects). Good database support for the fitness join alleviates these programs from using dedicated data structures and heuristic that makes re-use of the code base near to impossible. To illustrate:

- **In image databases** there is a need to compare high-dimensional data elements, such as color histograms under a distance metric[62]. A high-dimensional data structure is called upon, which is known to bring the dimensionality curse, while at the same time its semantic interpretation falls in the trap indicated for \( Q_3 \). A more precise fitness function specification may provide clues to weed out bad pairs by exploitation of the mathematical properties.

- **In image processing** there is a need to recognize complex-objects from primitive objects, e.g. triangularizations, grid-based decompositions, and more general image blobs. For example, lines should be recovered from line segments recognized during poor scanning [79, 81, 77]. The fitness function in this context binds segments that are angular similar, not too far apart, and of similar thickness. The clustering algorithm then attempts to maximize fitness to determine the candidates to pair into larger units.

- **In geographical information systems**, there is need to better support spatial joins and nearest neighbor search. To deal with it in an effective way emphasis is placed on spatial index structures, such as illustrated by [96, 42]. The fitness function in this context takes the form of an Euclidean distance metric using (a posteriori) filtering of candidate using attribute constraints.
6.3. **FITNESS JOIN ALGORITHMS**

- In *engineering* there is a need to deal with the imprecision of signals, calling for relaxation of the traditional equi-join condition. The concept of band-joins, e.g. \( r.age - d_1 \leq s.age \leq r.age + d_2 \), is a step into this direction [34], but also a special case of the fitness join.

- In *data mining* applications partial functions are used to classify information into coarse grain groups before the mining algorithms are fired to infer business models [54, 88] regrouping is needed. The groups depend on a fitness function.

- In *time series* applications, the fitness function translates into selecting a 'best-fit' between time series fragments [30, 29].

Although this list is by no means exhaustive, it highlights the need to consider specific support of the fitness join at the DBMS system level. Leaving the problem with the application programmer to deal with it on a case by case basis is from the database perspective not acceptable. We should find and assess algorithms and techniques to better support this large community.

## 6.3 Fitness Join Algorithms

In this section we analyze the necessity to extend a database kernel with a tailored implementation of the fitness joins. Such an extension should be weighted against required orthogonality of its instruction set and the opportunities to provide optimized versions otherwise not available or easily detectable by the query optimizer.

### 6.3.1 SQL Framework

From a computational perspective, the fitness joins can be readily supported by object-relational and O-O systems. However, due to the fitness join definition we have to fully exploit the DBMS's capabilities to group objects, to express the complex mathematical function (through a stored procedure), and to select specific elements from each group considered.

In the context of flat relational systems fitness joins lead to complex batches of SQL requests. Although multiple query optimization schemes may reduce the overhead incurred to some extent, the state-of-the-art in this field does not provide generic solutions in the near future.

To illustrate, in a relational system we can simulate the fitness join \( Q_1 \) using the SQL framework below. Its proper evaluation requires a nested query and exploits SQL's iterative semantics. Such queries are known to be notoriously difficult to optimize[1]. At best the optimizer can extract common sub-expressions or (erroneously) flatten it to a double cross-product expression.
SELECT r.name, s.name
FROM Person r, Person s
WHERE r.gender <> s.gender
AND abs(r.age - s.age) <= (SELECT abs(r.age - t.age) FROM Person
WHERE s.name <> t.name
AND r.gender <> t.gender)

Queries $Q_2$ and $Q_3$ are further complicated by a metric over set-valued attributes. Resolving this query in a flat relational framework is cumbersome. For an object-based framework with set operators it can be solved using a query to derive the table $\text{temp}(\text{Name}, \text{Name}, \text{Fitness})$. Subsequently, a simply aggregate query can extract the best partner. Traditional query optimizers have a hard time to optimize the cross-products and aggregates.

Query $Q_4$ uses the Euclidean distance metric, which is ideally supported by a Data Cartridge or Datablade for geographical information systems. The resulting basic SQL framework is shown below. Again not much can be done to improve response time. An R-tree index helps to solve point, region, and spatial joins, but it can not directly be used by a query optimizer to solve this fitness equation. The optimizer would have to detect that there is a better alternative for calling the two distance functions in the first place.

```
SELECT r.name, s.name
FROM Person r, Person s
WHERE r.gender <> s.gender
AND distance(r.age, s.age) <= (SELECT distance(r.age, t.age) FROM Person t
WHERE t.name <> s.name
AND r.gender = t.gender)
```

Note that this query depends on explicit naming of the distance function. Replacing it with the underlying definition would seriously jeopardize performance, because a query optimizer would not recognize easily the benefits of an R-tree. Likewise, the user could solve query $Q_4$ using a built-in function to access a nearest neighbor for any given point in the space covered. This solution works if, a priori, we split the Person into two tables, one for each gender, and to built a R-tree accelerator for fast access on locality. Thereafter, we can solve $Q_4$ with the following SQL query:

```
SELECT r.name, nearest Neighbor(Females, r.x, r.y)
FROM Males r
```

A query optimizer will typically generate a scan over Males and call the function for each instance. It will (normally) not consider building a search accelerator on Males first, followed by an index specific nearest neighbor algorithm, e.g. traversing the R-tree of both operands in parallel.
### 6.3. FITNESS JOIN ALGORITHMS

#### Figure 6.3: The Monet Ballroom database

<table>
<thead>
<tr>
<th>Person_name</th>
<th>Person_age</th>
</tr>
</thead>
<tbody>
<tr>
<td>oid</td>
<td>name</td>
</tr>
<tr>
<td>0</td>
<td>Tina</td>
</tr>
<tr>
<td>1</td>
<td>Brian</td>
</tr>
<tr>
<td>2</td>
<td>Mary</td>
</tr>
<tr>
<td>3</td>
<td>Susan</td>
</tr>
<tr>
<td>4</td>
<td>Edward</td>
</tr>
<tr>
<td>5</td>
<td>Owen</td>
</tr>
<tr>
<td>6</td>
<td>Ruth</td>
</tr>
<tr>
<td>7</td>
<td>Peter</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Person_gender</th>
<th>Person_x,y</th>
<th>Person_dance</th>
</tr>
</thead>
<tbody>
<tr>
<td>oid</td>
<td>name</td>
<td>oid</td>
</tr>
<tr>
<td>0</td>
<td>female</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>male</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>female</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>female</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>male</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>male</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>female</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>male</td>
<td>7</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Person_experience</th>
<th>oid</th>
<th>experience</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td></td>
</tr>
</tbody>
</table>

**6.3.2 Monet solutions**

To improve upon the situation sketched, we study the opportunities of extending a relational algebra engine. The system being considered here is Monet, a binary relational algebra engine, including powerful grouping primitives and facilities to extend its behavior through dynamic loadable modules [9, 12]. The Monet schema for the Ballroom database is shown in Figure 6.3.2.

The naive solution for the fitness joins \( R < f S \) is to generate an iterative program in the Monet Interface Language (MIL) [9], that collects the minima for each \( r \in R \). It would be the default route taken by most relational query optimizers.

In the remaining sections we focus on a more general scheme, using the fitness function of \( Q_1 \) as the focal point. Support for set-based operations

---

1 The papers can be accessed through http://www.cwi.nl/~monet
at the kernel level has been dealt with in the context of datamining support [11]. A synopsis is beyond the scope of this chapter. Furthermore, the spatial operators and their implementation in the context of Monet are reported in [13].

**Cross Table Solution**

The next step is to identify the necessary extensions to the relational algebra, such that performance gains can be obtained from bulk operations. In this quest, we follow Monet's approach to fully materialize results of binary operators. This has proven to be effective in most situations, because it trades storage space against memory cycles lost by MIL interpretation and context switching [12, 10]. The key problems to be addressed for the fitness joins are then to build a cross-table (sparse matrix) of the candidate pairs, calculate the fitness value per element, and retain the minima per row \((R)\) order.

The first step is to extend the algebra with the notion of cross table together with primitives to query it. In Monet the cross table could be represented by two binary tables \(R\text{map} : idx \rightarrow oid\) and \(S\text{map} : idx \rightarrow oid\) where their \(idx\) constitutes an index into the 2-dimensional space spanned by \(|R| \times |S|\). The OIDs can be used to access the attributes of \(R\) and \(S\) tuples, respectively. These tables are produced with a tagging operator, \(R.\text{tag}(X,Y,Z)\), which assigns the index value to each element in \(R \times S\) using a conventional array-layout algorithm as shown in Figure 6.4. The \(\text{tag}\)—operation can be implemented cheaply using a single scan over its operand.

Subsequently, cross table specific operators can be introduced, such as fetching elements using a \(idx\), slicing a portion of the cross table for further processing, transformation of the cross table, and other matrix-like operations. For solving the fitness problem we need cross-table aggregates such as \(\text{rowMin}, \text{rowMax}, \ldots\) to work on rows and columns. These are straightforward extensions of their regular implementations.

With the algebraic extensions in place, we can solve the fitness join query \(Q_1\) in the Monet Intermediate Language (MIL) as follows.\(^2\)

\(^2\) See for details on MIL [9] and http://www.cwi.nl/~monet
6.3. FITNESS JOIN ALGORITHMS

```
proc tag (R,X,Y,Z) := {
  var answer := new(int,int);
  var i :=0;
  var k :=0;
  R@batloop()
  var j:=0;
  i:= k;
  while(j<X) {
    answer.insert(i,$h);
    j:= j+1; i:= i+Y;
  }
  k:= k+Z;
  return answer;
}
```

Figure 6.4: Monet Tagging Operation

| $R \bowtie_f S = \{(r,s)|r \in R, s \in S \forall t \in S | f(r,s) \leq f(r,t)\} $ |
| --- |
| 0 R := gender.select("male") # R:oid → str |
| 1 S := gender.select("female") # S: oid → str |
| 2 Rmap := R.tag(count(S),1,count(S)) # Rmap:idx → oid |
| 3 Smap := S.tag(count(R),count(R),1) # Smap:idx → oid |
| 4 Rage := Rmap.join(age) # Rage:idx → int |
| 5 Sage := Smap.join(age) # Sage:idx → int |
| 6 delta := abs([Rage]-[Sage]) # delta: idx → int |
| 7 best := delta.minRow(count(S)) # best: idx → int |
| 8 pairs := best.duplicate() # pairs:idx → idx |
| 9 v1 := Rmap.reverse.join(pairs) # v1:oid → idx |
| 10 answer := v1.join(Smap) # answer:oid → oid |
| 11 a1 := R.reverse.join(answer) # a1:name → oid |
| 12 a2 := a1.join(S) # a2:name → name |

This script uses the Monet core algebra, multi-cast, and cross-table group operations. Lines 0,1 dissects the gender (binary) table into one for males ($R$) and females ($S$). Lines 2,3 constructs the candidate pairs mapping both tables into the cross-table space using the $tag$ operation. Lines 4,5 associates the age values with each cell.

The multi-cast expression $[abs](Rage-|Sage)$ in line 6 evaluates the age subtraction against all elements $Rage$ and $Sage$ with corresponding keys producing the table $\{Rage.idx,Rage.age - Sage.age\}$. Subsequently, the $abs()$ function is applied to all tuple tails, producing the table $\{Rage.idx,abs(Rage.age - Sage.age)\}$.

The minimum value per row is retained in line 7 using the cross-table enhancement. Line 8 replicates the key to prepare for the selection of couples.
oids in line 9 and 10. Finally, line 11 replaces the oid by the person’s name.

The storage required for this script to work is $|R| + |S| + 3 \times |R| \times |S|$ using eager garbage collection of intermediates. The processing cost is in the order of $4 \times |R| + 4 \times |S| + 5 \times |R| \times |S|$ steps. Evidently still too expensive to consider as the default for evaluation for large ballroom contests, because this algorithm still uses complete cross-products. Fortunately, for a large class of fitness functions a better solution exists.

**Bounded Theta-Join Algorithm**

Both the iterator and cross-table solutions ignored the mathematical properties of the fitness function. Yet, exploitation of these semantics may prove valuable in reducing the processing cost even further. For example, reconsider query $Q_1$, the minimum age difference, where the following equations hold:

$$\text{abs}(r.age - s.age) \leq \text{abs}(r.age - t.age) \iff \begin{cases} r.age \leq s.age \leq t.age \\ r.age \geq s.age \geq t.age \\ s.age \leq r.age \leq t.age \end{cases}$$

They illustrate that ordering on the age domain of $s$ and $t$ could be used to reduce the number of candidates. Furthermore, a pair $(r, s)$ such that $r.age \leq s.age$, can be determined with a theta-join. The algorithm could be rephrased accordingly and it halves the space of candidates considered. However, this is still too many, because the fitness join requires for each $r$ just one $s$; its closest neighbor under the fitness expression.

To tackle this problem, we have extended the relational algebra with a **bounded theta join** operation, e.g. $btj(R, S, \theta, n)$ where $\theta$ is one of the relational operators ($<, >, \leq, \geq$) and $n$ is the bound on the number of results retained per left operand value. An efficient implementation of the theta-join is already available in the Monet engine. It uses an index on one of the operands to speed up the search. If necessary, this index is created on the fly. Looping through the second operand, it produces the qualifying pairs, which are copied into the result table.

Its refinement for the bounded case $n = 1$ is relatively straightforward. It merely has to check the result table for duplicate insertion on the $r$ component and to retain the minimum $s$ encountered. The general case $n > 1$ requires slightly more work to retain the $n$–set of minimal values. The easiest way is to also create a sorted access path on the join attribute of $S$ (if not already there). Then we can optimize the $btj$ by merely doing an index lookup followed by a constrained ($n$)-step walk along the index to find the candidates of interest. Such tactical decisions are commonplace in the Monet kernel and prove to be highly efficient [9].

Using the bounded theta-join the Monet script of can be augmented as follows.
6.4. QUERY OPTIMIZATION SCHEMES

<table>
<thead>
<tr>
<th>$R \triangleleft_f S = {(r,s)</th>
<th>r \in R, s \in S \forall t \in S f(r,s) \leq f(r,t)}</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td><code>R := gender.select(&quot;male&quot;)</code> # <code>R:oid -&gt; str</code></td>
</tr>
<tr>
<td>1</td>
<td><code>S := gender.select(&quot;female&quot;)</code> # <code>S: oid -&gt; str</code></td>
</tr>
<tr>
<td>2</td>
<td><code>Rage := age.semijoin(R)</code> # <code>Rage:oid -&gt; int</code></td>
</tr>
<tr>
<td>3</td>
<td><code>Sage := age.semijoin(S)</code> # <code>Sage:oid -&gt; int</code></td>
</tr>
<tr>
<td>4</td>
<td><code>right := thetajoin(Rage,Sage,\leq,1)</code> # <code>right: oid -&gt; oid</code></td>
</tr>
<tr>
<td>5</td>
<td><code>left := thetajoin(Rage,Sage,\geq,1)</code> # <code>left: oid -&gt; oid</code></td>
</tr>
<tr>
<td>6</td>
<td><code>cand := union(right, left)</code> # <code>cand: oid -&gt; oid</code></td>
</tr>
<tr>
<td>7</td>
<td><code>Rmap := cand.mark()</code> # <code>Rmap: idx -&gt; oid</code></td>
</tr>
<tr>
<td>8</td>
<td><code>Smap := cand.reverse.mark()</code> # <code>Rmap: idx -&gt; oid</code></td>
</tr>
<tr>
<td>9</td>
<td><code>Rage := Rmap.join(Ra)</code> # <code>Rmap: idx -&gt; oid</code></td>
</tr>
<tr>
<td>10</td>
<td><code>Sage := cand.reverse.mark()</code> # <code>Rmap: idx -&gt; oid</code></td>
</tr>
<tr>
<td>11</td>
<td>`delta := {</td>
</tr>
<tr>
<td>12</td>
<td><code>best := delta.minRow(count(S))</code> # <code>best: idx -&gt; int</code></td>
</tr>
<tr>
<td>13</td>
<td><code>pairs := best.duplicate()</code> # <code>pairs:idx -&gt; idx</code></td>
</tr>
<tr>
<td>14</td>
<td><code>v1 := Rmap.reverse.join(pairs)</code> # <code>v1:oid -&gt; idx</code></td>
</tr>
<tr>
<td>15</td>
<td><code>answer := v1.join(Smap)</code> # <code>answer:oid -&gt; oid</code></td>
</tr>
<tr>
<td>16</td>
<td><code>a1 := R.reverse.join(answer)</code> # <code>a1:name -&gt; oid</code></td>
</tr>
<tr>
<td>17</td>
<td><code>a2 := a1.join(S)</code> # <code>a2:name -&gt; name</code></td>
</tr>
</tbody>
</table>

Lines 0,1 again identifies the males and females and line 2,3 obtain their age attribute. The left and right candidates are obtained with the bounded theta join in lines 4 and 5. They are combined to form the candidates in line 6. Line 7 and 8 construct the cross-table representation by introducing the cell identifier using a built-in routine `number()`. The remainder of the algorithm is identical to the previous version.

The prime effect of this preparatory step is that the storage cost is significantly reduced. Instead of $|R| \times |S|$ candidates there are only $2 \times |R|$ candidates in the main part of the algorithm. The processing cost is reduced accordingly to $2 \times |R| \times 5 + 2 \times |R|$.

6.4 Query Optimization Schemes

In this section we indicate the opportunities to exploit the fitness functions at query optimization time and accelerator data structures to support complex cases.

6.4.1 Mathematical Query Optimization

The fitness join expressions indicates a route for further exploration. For, when we compose functional expression over the operator set `{+, -, *, /, log, exp, sqrt}`, a single attribute, and constants, we maintain monotonicity of the result. This means that the bounded-theta-join solution presented in Section 7 can readily be applied.
The case considered for fitness function \( f_1 \) illustrates how a query optimizer can deal with discontinuous single attribute functions. It should break the underlying domains into pieces, such that within each piece the bounded-theta-join becomes applicable. Actually the optimizer should produce the first derivative in each point and determine the domain ranges where it can use the \( btj \) method. This analysis is relatively straightforward for the operator set considered.

Beyond these cases, a symbolic analysis of the fitness function could explore the following routes:

- **Sandwich method.** A fitness expression can be approximated using bounding functions that may be easier to compute, i.e.
  \[
  \hat{f}(r,s) \leq f(r,s) \leq \tilde{f}(r,s)
  \]

- **Transformation method.** If the fitness expression can be subjected to an affine transformation (rotation, translation) with clear boundaries, we could solve the function in a fraction of the domain and use a lookup table to speed up evaluation. For example,
  \[
  f(r,s) = f(r+\delta,s+\delta)
  \]

- **Candidate method.** If the operands involved 'identical objects' with respect to the attributes of concern in the fitness expression, it suffice to use one in solving the fitness problem.

The sandwich method is of interest if the underlying domain deals with sequence data. For example, in a stock exchange time series the band in which the stock price function fluctuates may be determined with a single scan over the underlying table. It provides a crude, but effective approximation of the function to filter candidates by looking at \( \hat{f}(r,s) \) (or \( \tilde{f}(r,s) \)) first. It is even possible to break the sequence into bull and bear market segments before fitness expressions are calculated.

The transformation method could be used in those cases where the underlying object incurs repetition. For example, in a fractal encoding of an image the fitness expression needs to be solved once for each fractal component. Subsequent join results can be found by applying the fractal transformation to the operands.

The candidate method calls for a projection over the attributes of interest and to keep the identity of one record \( r \) (or \( s \)) to partake in the fitness test. We then know that the result obtained holds for all other members in the same group.

The price paid for such optimizations is to include a mathematics reasoning system as part of the query optimizer. For elementary analysis this is not more difficult than finding common subexpressions. Finding appropriate sandwich functions could be encoded as static optimization rules for the classes of operators considered.
6.4.2 Data structures for fitness joins

Once we enter the realm of multi-attribute expressions, e.g. the distance fitness \( f_4 \), or set-based expressions \( f_2 \) the bounded-theta-join solution should be generalized to multiple dimensions. For example, the bounded-theta-join with distance 1 over a spatial domain equates with the nearest neighbor operation often deployed.

The set-based expressions can benefit from the partial order of the subset relation to reduce the number of candidates. Searching for the sets with maximum overlap, as defined by \( f_2 \), can be answered using the po-tree induced from this partial order. The bounded theta join can then use the order to speed-up matching as follows. To find pair \((r, s)\) maximizing \( f_2(r, s) \) we start from the set closest to the repertoire of \( r \). This starting point can be found in a single walk through the po-tree. A bounded number of sets with maximum overlap could then be determined by traversing the po-tree from this point in sub and super-set direction. Each newly found set should, if the bound is not reached, also be used as a new start point for traversal.

6.5 Evaluation

To assess the impact of the techniques described, we set out for a first experimental validation of the bounded-theta-join. For this purpose, we have implemented the ballroom contest in Monet. This system provides sufficient hooks to extend the algebraic engine with new operators and search accelerators. The implementation involved coding the naive, cross-table, and bounded-theta-join approach using a C-module. Moreover, we added a module for the po-tree over sets.

6.5.1 Dance partners by age

To evaluate the performance results, we faked a large ballroom contest with a party of trolls, dwarfs and elves to obtain a sizeable collection of different ages. The database was initialized with a equal number of males and females. We experimented with subsets in the range of 8 to 9192 dance couples. The dance couples are formed using query \( Q_1 \).

This choice assumes that traditional optimization steps, such as reducing the number of individual candidates as quickly as possible using attribute based selection, has already been performed. The second optimization step assumed, is to solve the fitness join for just one distinctive element in each group. Therefore, we project the operands over the attributes mentioned in the fitness join expression, keeping one person for the evaluation. After the fitness join has been evaluated, the equivalent persons can be joined back into the result to find for each male the group of females of the same minimal age.
The performance of the three algorithms for query $Q_1$ is shown in Figure 6.5(a). The incremental memory requirements (Kb) of the algorithms are shown in Figure 6.5(b).

![Figure 6.5: Execution Times and Memory Requirements.](image)

The experiments confirmed the expected behavior. The iterative solution is relatively fast for small dance parties (up to 64). Thereafter, the quadratic complexity of the algorithm results in poor performance. The Cross table solution performs even worse, because it consumes large amounts of memory, but also its performance follows the space consumption. Above 2048 dance couples the memory requirements even exceeded the available resources. The bounded-theta-join stands out as a winner, despite the overhead incurred in construction of an accelerator on the fly. The investment is quickly earned back in improved response time. The bounded-theta-join only needs to construct an accelerator when none of the two given tables is sorted or already contains an index structure. If the accelerators are available up front, the performance is even much better.

### 6.5.2 Dance partners by repertoire match

In a second dance contest problem couples were formed using a query $Q_2$: find the partner of opposite gender and with best overlap in dance repertoire. Again the operands are trimmed using the rules applied above. In the next step the search for the best overlapping dance repertoire is made. using
function \( f_2 \) to compare partners abilities.

Query \( Q_2 \) was solved using three algorithms again; iterator, cross table and \( po \)-tree. All three use bit sets to represent the repertoire. Their performance is shown in figure 6.6(a). The incremental memory requirements (Kb) of the algorithms are shown in figure 6.6(b).

![Figure 6.6: Execution Times and Memory Requirements.](image)

Again these experiments confirm the expected behavior. The iterator solution is optimal only for very small dance parties. Even smaller than for query \( Q_1 \). This stems from the more expensive \( f_2 \) for finding the set overlap. The cross table suffers even more from memory consumption, because more intermediate results are needed to find the set overlap. The \( po \)-tree is a clear winner, especially for larger dance parties. The investments of building it on the fly are quickly earned back. The tree size is only related to the number of dances involved; it is unrelated to the number of dancers. Therefore the cost of building the \( po \)-tree becomes even a relatively smaller investment for larger dance parties. Even better performance can be obtained when the \( po \)-tree is built up front.

### 6.6 Conclusion

In this chapter we have introduced the class of fitness joins, which appear regularly as building blocks in advanced database applications. They extend the traditional equi-, theta- and set-joins by a mathematical complex formula in the join condition combined with a selection from a (ranked) group.

We have shown that this class can be handled efficiently for relatively simple fitness functions using moderate extensions to the relational algebra. A method to manipulate cross-tables or sparse matrices provides the hook to represent results from sparsely populated cross-products. Furthermore, the bounded theta-join appears a valuable addition to the standard repertoire of algebraic operators and it can be implemented using traditional optimization
techniques. It extends early work on theta-joins [34] by uncovering the real handle to tackle the problem efficiently. Namely, judicious use of the monotonicity properties of compound mathematical functions combined with a variation of the theta-join.

Further optimization by exploring the mathematical properties of the fitness expressions have been indicated. It is an open research area, while most interactions with a database system come from applications where mathematical analysis is an integral activity. The optimizer framework of Monet is extended along this line and we plan to isolate and include the primitives for further experimentation in its algebra. Better support for fitness-based ranking remains on our wish list as well.