MONDRIAN: Annotating and querying databases through colors and blocks

Floris Geerts Anastasios Kementsietsidis

School of Informatics
University of Edinburgh
{fgeerts, akements}@inf.ed.ac.uk

Diego Milano*

Dip. di Informatica e Sistemistica
Università di Roma "La Sapienza"
diego.milano@dis.uniroma1.it

Abstract

Annotations play a central role in the curation of scientific databases. Despite their importance, data formats and schemas are not designed to manage the increasing variety of annotations. Moreover, DBMS's often lack support for storing and querying annotations. Furthermore, annotations and data are only loosely coupled. This paper introduces an annotation-oriented data model for the manipulation and querying of both data and annotations. In particular, the model allows for the specification of annotations on sets of values and for effectively querying the information on their association. We use the concept of *block* to represent an annotated set of values. Different *colors* applied to the blocks represent different annotations. We introduce a color query language for our model and prove it to be *complete* on the class of color (annotated) databases. We present MONDRIAN, a prototype implementing the proposed annotation mechanism, and we conduct experiments that investigate the set of parameters which influence the evaluation cost for color queries.

1 Introduction

From biology to astronomy, scientific databases play a central role in the advancement of science by providing access to large collections of data. At the same time, these databases are of particular interest to computer scientists due to the challenges that they pose in terms of data management [8]. Apart from the often staggering amounts of data, there are two additional characteristics of scientific databases that make their management challenging. First, the data stored in scientific databases often have different formats which range from flat-formatted files to images and electronic publications. Thus, part of the challenge is to *integrate* [13], *annotate* [7] and *cross-reference* [17] such diverse collections of data. Second, scientists often analyze data collected from a variety of sources and, in turn, the results of this analysis are used by others in a continuous feedback of (result) data. In such a setting, it becomes difficult for scientists to keep track of where the particular data that they are using came from. Thus, the challenge here is to maintain *data provenance* [9].

The objective of this paper is to offer a new model to annotate scientific databases (although the model can be applicable in other contexts). In the following paragraphs, we review some of the key issues we want to address. We present these issues through examples drawn from the domain of biological databases. Figure 1 shows example relations from three biological sources, namely, GDB [2] (a database about human genes), Swissprot [3] (a database about proteins), and PIR [1] (a database about protein sequences). In Figure 1(a), relation GDB records for each gene, its identifier, name, and chromosome. In Figure 1(b), relation Swissprot stores the identifier of each protein, its name, and the related species. Finally, in Figure 1(c), the PIR relation stores protein identifiers along with their names.

* Research done while visiting University of Edinburgh

gid	gname	chr
120231	NF1	17
120232	NF2	22
120233	NGFB	1
120234	NGFR	17
120235	NHS	21

(a) GDB relation

sid	sname	origin
P01138	Nerve growth factor	Human
P08138	TNR16	Human
P14543	Nidogen	Human
P21359	Neurofibromin	Human
P35240	Merlin	Human

(b) Swissprot relation

pid	pname
A01399	Nerve growth factor
A25218	Tumor necrosis factor
A45770	Merlin
I78852	Neurofibromatosis
Q6T45	Nancy-Horan syndrome

(c) PIR relation

Figure 1: Relations from three biological sources

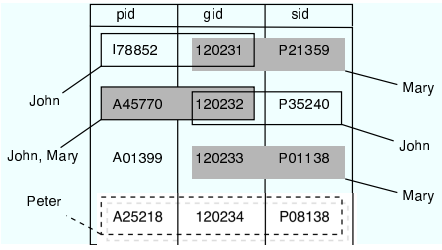


Figure 2: An integrated relation

We make two key claims about annotations. First, we argue that for an annotation mechanism to be useful in practice, it should be able to support the annotation of *value associations*. Existing annotation mechanisms assume that each annotation is attached to a particular value of a specific attribute (e.g., see [6]). So, we can annotate, for example, the gene name *NF1*, in Figure 1(a), with the name of the researcher that discovered this gene. A possible implementation of such a mechanism is to add one more column in the relation to record such an annotation. So in our example, we can add column *annot_gname* in the relation of Figure 1(a) and use this column to store the annotations of the *gname* column values.

However, in a number of domains, including scientific databases, it becomes increasingly important to annotate not only data values but also value associations. For example, consider the relation shown in Figure 2. This relation results by partially integrating the relations in Figure 1. The relation associates gene identifiers from the GDB database to protein (and protein sequence) identifiers from the Swissprot and PIR databases. The semantics of this association is that the specified gene is related to the indicated protein (and protein sequence). Such relations are widely used in the biological domain and offer a quick way to cross-reference and establish associations between independent biological sources [17]. In such a setting, it is important to record, for each integrated tuple, what evidence exists about a particular association. In the figure, we show possible annotations of the integrated relation in the form of blocks and block labels. In more detail, blocks are used to indicate the set of values for which an annotation exists, while block labels are used to indicate the annotations themselves. In our example, the annotations indicate the names of the curators who verified that a particular association holds. For example, in the first tuple, a block indicates Mary's belief that the gene with GDB id *120231* produces protein with id *P21359*. This type of annotation is a statement about neither the gene nor the protein, but rather about their relationship. One can think of using such a mechanism to annotate the *relationship* between any two (or more) entities. In comparison, existing mechanisms are capable of only annotating the entities themselves. We note that to represent annotations such as the ones described here, it does not suffice to add an annotation column for each attribute of the relation. One objective of this paper is to propose a suitable representation so that our annotation mechanism can be implemented in practice.

Our second claim about annotations is that they should be treated as first-class citizens of the database, that is, we should be able to query values and annotations alike (in isolation or in unison). Currently, query languages operate only on data values and the corresponding annotations are

propagated accordingly in the query result (e.g., see [6]). However, for curators, annotations are of equal and sometimes of greater importance than values. A curator using the relation in Figure 2 might want to find which tuples are annotated by either John or Mary. This is still a value-based query, but it refers to an annotation value which, as the figure shows, spans attribute boundaries, and it can appear over distinct attribute sets, in different tuples. As another example, the curator might be interested in finding which gene-protein sequence (*gid, sid*) pairs are annotated, and by whom. This is not a value-based annotation query. Rather, it refers to the schema on which annotations are applied for each tuple. As a last example, we note that sometimes the lack of annotations might also be of interest to a curator. In a heavily curated database, like Swissprot, a curator might want to find which gene-protein (*gid, pid*) pairs are not annotated. All these operations assume that we have a query language capable of expressing queries over annotated databases.

Desirable properties of such a query language are that it is independent of the chosen representation of annotations and that it is user friendly. Any relational representation of annotated databases offers the relational algebra or, on a practical level, SQL as candidate query languages. However, a natural requirement is that each query posed in these languages should be *annotation-aware*. Furthermore, the result of the query must be interpretable as an annotated database again. It is clear that one needs to pose severe syntactic and semantic conditions on such queries in order to achieve this goal. It is not clear what these conditions should be.

We opt for a different approach and introduce a new query language which we will refer to as the *color algebra* (since we use colors to represent annotations). By definition, any query in this language produces a colored (annotated) relation on every input colored relation. Moreover, the semantics of colors and blocks is transparent in each operator. This facilitates the manipulation and querying of colored databases. In particular, the queries described above are easily expressed in the color algebra.

The contributions of this paper are as follows:

• We introduce an annotation mechanism for relational databases that is capable of annotating both single values and the associations between multiple values. We investigate generic properties of annotations and, through these properties, we are able to define annotations with a range of semantics. To the best of our knowledge, our mechanism is both the first to support the annotation of value associations and the first to investigate the properties of annotations.

• We introduce an algebra that can be used to express queries on values and annotations alike. Our algebra supports well-known algebra operators, like selection and projection, along with new operators that are particular to the querying of annotations. We formalize the notion of annotation in relational databases and we show that the introduced algebra is complete on the class of annotated relational databases.

• We present MONDRIAN[1] which is an implementation of our annotation mechanism over a relational DBMS. We investigate the space overhead of our representation, and we study the cost of evaluating queries over annotated databases and the parameters that influence this cost.

The remainder of this paper is organized as follows. First, we review related work. Then, in Section 2, we introduce the basic notions of colors and blocks. Section 3 presents the color algebra while Section 4 describes the relational representation and presents the completeness result. Section 5 introduces the MONDRIAN system and offers a description of our implementation and experiments. The paper concludes in Section 6 with a summary of the results and a discussion on future work.

1.1 Related Work

Most existing annotation systems focus on text and HTML documents (e.g., Annotea [16]) and are often specialized to support annotations for a particular kind of data, e.g., genomic sequences [12, 7]. Research on these systems has been focused on scalability, distributive support of annotations, and other features.

[1] Piet Mondrian: Dutch painter whose paintings mainly consist of colored blocks.

In the relational setting, Bhagwat et al. [6] recently proposed an annotation management system for relational databases in which relational data can be annotated. More specifically, they store annotations in special attributes and extend the Select-Project-Join-Union fragment of SQL with a PROPAGATE clause which allows the user to specify how annotations should propagate. The focus is thus on the propagation of the annotations through queries, and the issue of how to query these annotations is not addressed. Also, only single values are annotated.

A more extensive literature exists regarding the computation of provenance. Annotations provide a solid way of keeping track of provenance. Indeed, computing provenance by forwarding annotations along data transformations has been proposed in various forms [19, 5, 16, 6, 18]. The data provenance problem without the use of annotation is studied by Cui et. al [11], Buneman et. al [9, 10], and Widom [20]. In this work, a "reverse" query is generated to compute data provenance. In this paper, we will not address the issue of provenance. Instead we provide a foundation on which both provenance information and other forms of annotations can be managed.

An unrelated work (although the title suggests otherwise) regards "Colorful XML" [15]. In this paper a new XML data model, referred to as multi-colored trees, is introduced. Colors are used to add semantic structure over the nodes in the XML data.

2 Colors and blocks

As already mentioned, our aim is to provide a mechanism for annotating groups of attribute values. We refer to such a group of attribute values as a *block*. As an example, in Figure 2, there are six different blocks, and each block has an associated annotation. In the remainder of the paper, for ease of presentation and notational convenience, we assume that each annotation is represented by a *color*. Therefore, instead of talking about annotations and annotated blocks we talk about *colors* and *colored blocks*, respectively. Similarly, we talk about *colored databases* (databases that are annotated) and *color queries* (queries on annotated databases) that are written using a *color algebra* (an algebra that accounts for annotations).

2.1 Coloring notations and properties

We first describe the data model used in this paper. Let $\mathbf{D}$ be a standard relational database consisting of the relations $R_1, \ldots, R_k$. For each relation R_i, we denote its set of attributes by $sort(R_i)$, while we use r_i to denote an instance of the relation. We use upper-case letters early in the alphabet $(A, B, \ldots)$ to denote attribute names while upper-case letters late in the alphabet $(X, Y, \ldots)$ are used to denote sets of attributes. Accordingly, lower-case letters early in the alphabet $(a, b, \ldots)$ are used to denote attribute values, while those late in the alphabet $(x, y, \ldots)$ are used to denote sets of attributes values. Finally, $\mathcal{C}$ denotes a set of colors.

Let r be an instance of relation R and let t be a tuple in r. The annotation, or *coloring*, of a tuple t is performed through a coloring function χ. Function χ accepts as input a tuple t and a non-empty set of attributes $Y \subseteq sort(R)$ and assigns a set of colors to the values in $t[Y]$. For a tuple t, the triplet $(t, Y, \chi(t, Y))$ defines a *color block* which consists of the attribute values in $t[Y]$ along with their assigned colors. If $\chi(t, Y) = \emptyset$, then the values in $t[Y]$ are not within a color block.

Example 1. *Consider the relation in Figure 2. Then, the coloring of each tuple in the relation is expressed through the following coloring function χ (where, t_i is the ith tuple in the relation):*

$\chi(t_1, \{pid, gid\}) = \{John\}$ $\quad$ $\chi(t_1, \{gid, sid\}) = \{Mary\}$
$\chi(t_2, \{pid, gid\}) = \{John,\ Mary\}$ $\quad$ $\chi(t_2, \{gid, sid\}) = \{John\}$
$\chi(t_3, \{gid, sid\}) = \{Mary\}$
$\chi(t_4, \{pid, gid, sid\}) = \{Peter\}$

Note that $\chi(t_i, Y)$ is equal to the empty set, for every tuple t_i and every set of attributes Y, other than the ones mentioned above. □

Color blocks permit the specification of annotations with diverse semantics. As an example, consider Figure 2. A color could represent a curator who has approved or verified an association of values. Alternatively, a color may indicate just the provenance of values. We present below a set of properties of colors and blocks that can be used to capture such diverse semantics. Furthermore, we use these properties while answering queries over the colored databases.

Block overlapping We allow an attribute value $t[A]$, $A \in sort(R)$ to participate in more than one color blocks. Intuitively, this property allows a value to participate in multiple independent annotated associations. This is the case in Figure 2 where, in the first tuple, the gene with GDB id *120231* is colored twice, once because it produces the Swissprot protein *P21359* and once because it is associated to the PIR protein sequence with id *I78852*.

Inheritance We say that a coloring function is *inheriting* if for every color block $(t, Y, \chi(t, Y))$, and every set of attributes $Y' \subseteq Y$, the following blocks are implied:

$$(t, Y', \chi(t, Y')) \text{ is a block with } \chi(t, Y') \supseteq \chi(t, Y)$$

Intuitively, in certain applications, we want to color a set of values by placing all of them within a block and, at the same time, we want any possible subsets of these values to inherit this color. For example, assume that the annotations in Figure 2 denote that certain genes and proteins are mentioned in a publication whose author name appears in the annotation. Then, we expect that the annotation is also inherited by any subset of these values since any such subset is in the publication.

We say that a coloring function is *non-inheriting* if no blocks are implied apart from the ones explicitly defined by the function. As we discussed in the introduction, we often annotate values because they have a property as a set, while no subset of these values has the property. As an example, consider the annotations in Figure 2 which represent names of curators who believe that certain genes and proteins are related. Remember that these annotations are on the relationship between genes and proteins and not the individual values.

Transitivity We say that a coloring function χ is *transitive* if for any two blocks $(t, X, \chi(t, X))$ and $(t, Y, \chi(t, Y))$, with $\chi(t, X) \cap \chi(t, Y) \neq \emptyset$, the following color block exists $(t, X \cup Y, \chi(t, X) \cap \chi(t, Y))$. Intuitively, a transitive function merges blocks whose sets of colors overlap. This might be desirable when, for example, we have a coloring function such as the one of Figure 2, which expresses the belief of a curator in an association. In the second tuple of the figure, curator John has validated the association of both the pid *A45770* with the gid *120232*, and the association of this gid with sid *P35240*. If we assume a transitive coloring function, then a color block with all three values is implied whose semantics is that John has validated the association of all the values in the tuple.

Given the introduced notation and properties, in the next section, we present an algebra that can be used to query colored databases. The algebra includes standard operators like selection (σ), projection (π), product ($\times$), renaming (ρ) and union ($\cup$), along with a set of operators that are particular to the manipulation of colors and colored blocks.

3 Color algebra (CA)

In what follows, we introduce the set of operators of the color algebra (CA). We start with two basic operators through which we can refer relations in our database and introduce new constants.

***Input relation*:** The operator R accepts as input an instance $\langle r, \chi \rangle$ and returns $\langle r, \chi \rangle$, if r is an instance of R, and the empty relation otherwise.

***Unary singleton constant*:** We allow for the creation of (un-)annotated single values. Specifically, the operator (A, a) takes as input any instance $\langle r, \chi \rangle$ and returns the instance of sort $\{A\}$ containing a unique element a. The operator (A, a, c) takes as input any instance $\langle r, \chi \rangle$ and returns the instance of sort $\{A\}$ containing a unique element a which is in a block of color c (or $\chi(a, A) = \{c\}$).

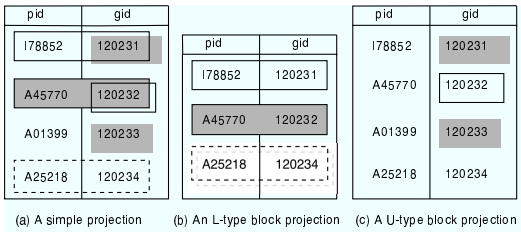


Figure 3: The projection operators

***Projection*:** We define the projection $\pi_{A_1 \cdots A_k}$ as the operator which takes as input any instance $\langle r, \chi \rangle$ of sort containing $\{A_1, \ldots, A_k\}$ and returns the instance $\langle r', \chi' \rangle$ of sort $\{A_1, \ldots, A_k\}$ such that

$$r' = \{t[A_1, \ldots, A_k] \mid t \in r\} \text{ (normal projection)}$$

and for any $t \in r$, and any $Y \subseteq \{A_1, \ldots, A_k\}$,

$$\chi'(t[A_1, \ldots, A_k], Y) = \bigcup_Z \chi(t, Y \cup Z),$$

where Z ranges over all subsets of $sort(R) \setminus \{A_1, \ldots, A_k\}$. Our projection operator treats the coloring function as an inheriting one since it projects the blocks in each tuple of r to the projected attributes. In the end of this section, we show how to define an alternative projection operator that treats the coloring function as non-inheriting.

Example 2. *Consider again the relation $\langle r, \chi \rangle$ shown in Figure 2. Then, expression $\pi_{pid,gid}(r)$ returns the relation r' shown in Figure 3(a). Notice that in tuples t_1, t_2 and t_3, the projection introduces blocks consisting only of attribute $\{gid\}$. Furthermore, in tuple t_4, it introduces a block consisting of attributes $\{pid,gid\}$.* □

The projection operator uses a schema constraint T to remove the parts of a relational instance which are uninteresting, or irrelevant, for the task at hand. It proves useful to offer a corresponding operator that works on the annotation level by removing annotations that do not satisfy certain schema constraints.

***Block Projection*:** We offer two types of block projection that allow for the projection of blocks based on whether blocks contain or are contained in a specified set of attributes. Specifically, the L-type (Lower) block projection operator $\Pi^L_{A_1,\ldots,A_k}$ takes as input an instance $\langle r, \chi \rangle$ of sort containing $\{A_1, \ldots, A_k\}$, and returns the instance $\langle r', \chi' \rangle$ of the same sort defined by

$$r' = \{t \mid t \in r \text{ and there exists a block}(t, Y, \chi(t, Y)) \text{ with } A_1, \ldots, A_k \subseteq Y\}$$

and for any $t \in r'$, and any set of attributes $Y \subseteq sort(R')$,

$$\chi'(t, Y) = \begin{cases} \chi(t, Y) & \text{if } \{A_1, \ldots, A_k\} \subseteq Y,\ \chi(t, Y) \neq \emptyset; \\ \emptyset & \text{otherwise.} \end{cases}$$

The U-type (Upper) projection operator $\Pi^U_{A_1,\ldots,A_\ell}$ is defined similarly, except that $r' = r$ and in the definition of $\chi'(t, Y)$, $Y \subseteq \{A_1, \ldots, A_k\}$ must hold. We also define $\Pi^L_\emptyset = \text{Id}$, while $\Pi^U_\emptyset$ only returns the unannotated tuples.

Example 3. *Consider the relation $\langle r, \chi \rangle$ in Figure 3(a). Assume that we want to find all the tuples with at least one annotation that involves the pid attribute. Then, we can use the expression $\Pi^L_{pid}(r)$. The expression returns the pair $\langle r', \chi' \rangle$, shown in Figure 3(b). Notice that tuples that do not have an annotation involving the pid attribute, and blocks that do not annotate the attribute, are removed.*

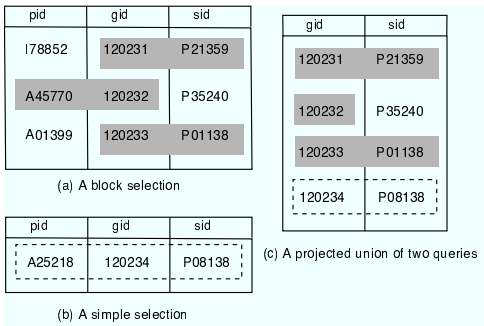


Figure 4: The selection and union operators

On the other hand, we may want all the tuples of relation r that might have an annotation involving only the gid attribute. Then, the expression $\Pi^U_{gid}(r)$ finds all such tuples. The resulting relation is shown in Figure 3(c). Notice that while the Π^L projects out unannotated tuples, the Π^U operator preserves them. This is because the former operator requires the existence of blocks, while the latter only specifies a maximum set of attributes that a block can include. □

***Selection*:** The operator $\sigma_{A=a}$ takes as input any instance $\langle r, \chi\rangle$ and returns the instance $\langle r', \chi'\rangle$ of the same sort defined by $r' = \{t \mid t \in r, t[A] = a\}$ and χ' is the restriction of χ to r'.

The selection operator $\sigma_{A=B}$ is defined over any instance $\langle r, \chi\rangle$ of sort containing $\{A, B\}$ and returns the instance $\langle r', \chi'\rangle$ of the same sort defined by $r' = \{t \mid t \in r, t[A] = t[B]\}$ and where χ' performs a similar selection on the blocks. More specifically, for any $t \in r'$, and any $Y \subseteq sort(R')$ we have that

$$\chi'(t, Y) = \begin{cases} \chi(t, Y) & A, B \notin Y; \\ \chi(t, Y) \cap \beta(t, A) \cap \beta(t, B) & \text{otherwise.} \end{cases}$$

where $\beta(t, A)$ (resp. $\beta(t, B)$) is the set of colors of all blocks in t containing attribute A (resp. B). So, for tuples that satisfy the selection condition at the value level, only those blocks containing A (resp. B) for which there exists a block, of the same color, containing B (resp. A), are selected. If no such blocks exist, the selection attributes become unannotated. Thus, selection requires an agreement on both the data and block level, as one of our following examples illustrates (Example 5).

Similar to the selection operator, which can be used to identify tuples that have a particular data value, it is desirable to offer a block selection operator to identify blocks that have a specific color.

***Block Selection*:** The operator Σ_c, where $c \in \mathcal{C}$, takes as input any instance $\langle r, \chi\rangle$ and returns the instance $\langle r', \chi'\rangle$ of the same sort defined by

$$r' = \{t \mid t \in r \text{ such that there exists a block in } t \text{ of color } c\},$$

and for any $t \in r'$ and any set of attributes $Y \subseteq sort(R)$, $\chi'(t, Y) = \chi(t, Y) \cap \{c\}$.

***Union*:** The union operator takes as input any two instances $\langle r, \chi_r\rangle$ and $\langle s, \chi_s\rangle$, of the same sort, and returns the instance $\langle r', \chi'\rangle$ of the same sort defined by $r' = s \cup r$ (set union) and for any $t \in r'$, and any set of attributes $Y \subseteq sort(R')$, we have that $\chi'(t, Y) = \chi_r(t, Y) \cup \chi_s(t, Y)$.

Example 4. *Consider again relation $\langle r, \chi\rangle$ in Figure 2. Assume that we want to find all the tuples that have a block annotated by Mary, or concern the protein with sid P08138. Also, assume that we are only interested in keeping the $\{gid, sid\}$ attributes from these tuples. Then, the expression*

$$\pi_{gid,sid}((\Sigma_{Mary}(r)) \cup (\sigma_{sid=\text{“}P08138\text{”}}(r)))$$

returns the desired result. Figure 4 shows both the intermediate results for each part of the union, and the final result of the query. Notice that the block selection operator maintains only the tuples that have at least one annotation from Mary. At the same time, from these tuples, the operator keeps only the blocks belonging to Mary. On the other hand, a selection predicate of the form $A = a$ focuses on values without altering the block structure. Our next example shows that this is not the case for selections of the form $A = B$.

The next two three operators are of particular importance both for the completeness of our algebra and because, along with the selection operator, they can be used to define the color join.

***Product*:** The product operator $\times$ takes as input any two instances $\langle r, \chi_r \rangle$ and $\langle s, \chi_s \rangle$ of disjoint sorts and returns the instance $\langle r', \chi' \rangle$ of sort $R' = sort(R) \cup sort(S)$ defined by $r' = r \times s$ (normal product) and for any tuple $t \in r'$ and set of attributes $Y \subseteq sort(R')$,

$$\chi'(t, Y) = \begin{cases} \chi_r(\pi_{sort(R)}(t), Y) & \text{if } Y \subseteq sort(R); \\ \chi_s(\pi_{sort(S)}(t), Y) & \text{if } Y \subseteq sort(S); \\ \emptyset & \text{otherwise.} \end{cases}$$

***Merge*:** Assuming a transitive coloring function, a natural operation on blocks is merging. The operator $merge_{Y,Z}$, with Y, Z being sets of attributes such that $Y \cap Z = \emptyset$, takes as input instances $\langle r, \chi \rangle$ of sort $sort(R)$ containing $Y \cup Z$ and returns the instance $\langle r', \chi' \rangle$ of the same sort defined by $r' = r$ and for any $t \in r'$ and any set of attributes $X \subseteq sort(R)$,

$$\chi'(t, X) = \chi(t, X_1) \cap \chi(t, X_2),$$

where $X = X_1 \cup X_2$, $X_1 \subseteq Y$, $X_2 \subseteq Z$ and $\chi(t, X) = \emptyset$. Intuitively, the merge operator considers each tuple t and it identifies pairs of blocks that are contained in Y and Z, respectively, and have the same color. Then, it replaces two equi-colored blocks with a new block that is the result of their merging. Blocks that are contained in Y and Z but cannot be merged, are dropped, as are the blocks not contained in Y and Z.

***Renaming*:** Let f be an attribute renaming of a finite set of attributes R such that $f(A) \neq A$. The renaming operator δ_f accepts as input an instance $\langle r, \chi \rangle$ of sort R, and returns the instance $\langle r', \chi' \rangle$ of sort $f(R)$ defined by

$$r' = \{t' \mid t \in r \text{ and } \forall A \in R, t(A) = t'(f(A))\},$$

Furthermore, for any $t' \in r'$ and any set of attributes $Y' \subseteq sort(R')$, $\chi'(t', Y') = \chi(t, Y)$, where $Y = \{A_1, \ldots, A_k\}$ and $f(A_i) = A'_i$ and $t'[f(A_i)] = t[A_i]$ for $i \in [1, k]$.

Example 5. *Consider relation $\langle r, \chi \rangle$ from Figure 3(b) and relation $\langle r', \chi' \rangle$ from Figure 4(c) (the relations are copied in Figures 5(a) and (b) for convenience). Figure 5(c) shows the relation that results from taking the product of r and r' and applying an equality condition on the gid attribute, that is,*

$$\sigma_{gid=gid'}(r \times \delta_f(r'))$$

where f is a renaming function such that $f(gid) = gid'$ and $f(sid) = sid'$. Notice that the selection only selects those blocks containing gid which have a equi-colored block containing gid', and vice versa. If no such blocks exists, as is the case for gid 120231, the resulting tuple is unannotated.

If we use projection to remove one of the two gid attributes, we end up with a different block structure, depending on which attribute we project out. The two situations are depicted in Figures 6(a) and (b). We can avoid such an undesirable situation by using the $merge_{\{pig,\ gid\},\{gid',\ sid'\}}$ operator. After the merge operator is applied, irrespectively of which of the two attributes is projected out, the resulting relation is the same. This is shown in Figure 6(c).

***Recolor*:** The recolor operator ρ takes as input any two instances $\langle s, \chi_s \rangle$ and $\langle r, \chi_r \rangle$ and returns the instance $\langle r', \chi' \rangle$ of sort $sort(R)$ defined by $r' = r$ and for any tuple $t \in r'$ and set of attributes

pid	gid
I78852	120231
A45770	120232
A25218	120234

(a) A relation

gid	sid
120231	P21359
120232	P35240
120233	P01138
120234	P08138

(b) Another relation

pid	gid	gid'	sid'
I78852	120231	120231	P21359
A45770	120232	120232	P35240
A25218	120234	120234	P08138

(c) Cartesian-product followed by selection

Figure 5: Selection and product operators

pid	gid	sid'
I78852	120231	P21359
A45770	120232	P35240
A25218	120234	P08138

(a) Projecting out gid'

pid	gid'	sid'
I78852	120231	P21359
A45770	120232	P35240
A25218	120234	P08138

(b) Projecting out gid

pid	gid'	sid'
I78852	120231	P21359
A45770	120232	P35240
A25218	120234	P08138

(c) Projection after the merge operator is applied

Figure 6: The merge operator

$Y \subseteq sort(R')$,

$$\chi'(t,Y) = \begin{cases} getcolor(\langle s, \chi_s \rangle) & \text{if } \chi_r(t,Y) \neq \emptyset; \\ \emptyset & \text{otherwise.} \end{cases}$$

Here $getcolor(\langle s, \chi_s \rangle)$ gets all the colors of all the blocks in $\langle s, \chi_s \rangle$.

This concludes the introduction of the basic operators of the positive color algebra (CA). Since the result of each operator on a colored relation is again a colored relation, we can compose all operators. Note that our algebra does not account for negation. Its consideration is part of our future work.

Definition 1. *The (positive) color algebra (CA) consist of all expressions obtained by composing a finite number of the operators introduced above.* □

Before we characterize the expressive power of this algebra, we make the following observations. The first observation concerns the color queries that are written using the color operators that have a relational algebra counter-part (e.g. selection, projection, product). Each such color query results in the same set of tuples as we would get by applying the relational algebra counterpart on an unannotated database. The difference between the two queries is the presence of blocks. Thus, by using the CA algebra, we don't *lose* any data.

Our second observation relates to our discussion on annotation properties. By default, we offer a projection operator that treats a coloring function as inheriting and a merge operator that assumes a transitive one. To offer additional flexibility, we choose not to encode these properties in each individual block. Instead, at query time, we allow each user to decide how she wants to interpret the coloring functions by applying appropriate operators. E.g., a projection which treats a coloring function as non-inheriting can be easily expressed in CA. Indeed, the expression $\pi_{A_1,\cdots,A_k}(\Pi^U_{A_1,\cdots,A_k}(r))$ does exactly this.

Our final observation is that we can define the *join* in CA as well. The join identifies attributes based on both their values and block structure and merges the "common" blocks. More specifically, we define $\langle r, \chi_r \rangle \bowtie \langle s, \chi_s \rangle$ by the expression (assuming that A_i and A_j are the attributes to join on)

$$\pi_{sort(r)\cup sort(s)\setminus\{A_j\}}\big(merge_{sort(r),sort(s)}(\Pi^L_{A_i}(\sigma_{A_i=A_j}(r \times s)) \cup (\Pi^L_{A_j}\sigma_{A_i=A_j}(r \times s)))\big)$$

As noted in Example 5, we obtain an equivalent expression when projection includes A_j instead of A_i.

4 Connection with relational model

4.1 Relational representation

In this section, we provide a relational representation of colored databases. In what follows, we define a mapping rep from colored databases to a special type of relational databases. Let $\mathcal{CD}$ denote the class of colored databases, and let $\mathcal{S}$ denote the image set $\mathsf{rep}(\mathcal{CD})$ in the class of all relational databases. We also define the inverse mapping rep^{-1}, but only on $\mathcal{S}$.

Let $\langle r, \chi \rangle$ be a colored relation instance over R. Let $sort(R) = \{A_1, \ldots, A_k\}$. We define the relation S of sort $\{A_1, \ldots, A_k, B_1, \ldots, B_k, \gamma\}$, where the attributes A_i are of type *data*, the attributes B_i are of type *Boolean*, and the γ attribute is of type *color*. Moreover, we assume that there is a bijection between the data and Boolean attributes. We denote by $assoc(A_i, B_i)$ that A_i are B_i are mapped onto each other by this bijection. The Boolean attributes are used to determine which of the corresponding data attributes belong to a block. We denote the class of relational databases satisfying the above schema constraints by $\mathcal{S}$.

For each annotated tuple $t \in r$ and each $Y \subseteq sort(R)$ such that $\chi(t,Y) \neq \emptyset$, we populate the relation $\mathsf{rep}(\langle r, \chi \rangle)$ by adding to it the set of tuples

$$\{(t, B_1, \ldots, B_k, c) \mid c \in \chi(t,Y)\},$$

where $B_i = 1$ if $A_i \in Y$ and $assoc(A_i, B_i)$ holds, and $b_i = 0$ otherwise. Furthermore, for each tuple $t \in r$, we insert an unannotated tuple $(t, 0, \ldots, 0, c)$ in the representation, where c is some arbitrary color. This concludes the definition of mapping rep on single colored relations. The extension to colored databases, i.e., a set of colored relations, is defined analogously.

Concerning rep^{-1}, if we are given a relational instance s in class $\mathcal{S}$, whose schema is $\{A_1, \ldots, A_k, B_1, \ldots, B_k, \gamma\}$, we can easily convert s back to a colored relation $\mathsf{rep}^{-1}(\langle r, \chi \rangle)$. Due to lack of space, we omit the details of how this is achieved.

Example 6. *Consider again the colored relation $\langle r, \chi \rangle$ given in Figure 2. The following relation contains the three tuples in $\mathsf{rep}(\langle r, \chi \rangle)$ which correspond to the representation of the first tuple in $\langle r, \chi \rangle$. We assume that $assoc(pid, bpid)$, $assoc(gid, bgid)$, and $assoc(sid, bsid)$.*

pid	*gid*	*sid*	*bpid*	*bgid*	*bsid*	γ
I78852	120231	*P21359*	0	0	0	*c*
I78852	120231	*P21359*	1	1	0	*John*
I78852	120231	*P21359*	0	1	1	*Mary*

where c is arbitrary color. The other tuples in $\mathsf{rep}(\langle r, \chi \rangle)$ are obtained similarly. □

A few words about the choice of representation. First, we note that we can normalize our representation so that the values of a tuple are not repeated for every block. Second, our experience shows that our representation offers significant savings, in terms of space, over alternative representations. Consider, for example, a representation where a column is created for each element in the power-set of the set of attributes of a relation. Assuming single-colored blocks, for each block in an annotated tuple, its color becomes the value of the column that corresponds to the set of attributes in the block. Such a representation has a schema which is exponential, in the size of the schema of the annotated relation, and it requires one tuple for each annotated tuple in the relation. Our representation has a schema of linear size but it requires one tuple for each color of each block. Thus, exponential number of tuples might be necessary to represent an annotated tuple. However, in our representation, apart from the unannotated tuples, a tuple is created only if a block exists. In the alternative representation, we have a column for each set of attributes, irrespectively of whether a block for this set exists, or not. Furthermore, our encoding of blocks through Boolean attributes offers additional space savings.

4.2 Expressiveness

The relational representation of colored databases suggests another candidate query language, namely the normal relational algebra on this representation and specifically the fragment consisting of the union of conjunctive queries. In this section, we establish a link between the CA algebra and this fragment of the normal relational algebra. In what follows, we only consider positive algebra queries.

4.2.1 Colored relational algebra (CRA)

It is clear that not every relational algebra query on a representation of a colored relation results in a representation of a colored relation again. For example, any projection consisting only of data attributes, does not correspond to a colored database. It would therefore be desirable to identify the class of algebra expressions which, when applied to any representation of a colored relation, results in a representation of a colored relation.

Recall the $\mathcal{S}$ is the set of relational databases which represent a colored database through the rep mapping.

Definition 2. A positive relational algebra query Q is *colored* if for every relational database $\mathbf{D} \in \mathcal{S}$, the query result $Q(\mathbf{D}) \in \mathcal{S}$ as well.

We identify the following three necessary and sufficient syntactic conditions for a positive relational algebra query to be colored. Without loss of generality we may assume that such query is given in normal form [4]. More specifically it is the union of conjunctive queries of the form $\pi_X \sigma_F(R_1 \times \cdots \times R_k)$. Clearly, a positive query is colored if it is the union of colored conjunctive queries. It is easily verified that a conjunctive query is colored iff it satisfies the following three properties: (i) the projection must contain a single color attribute, (ii) since each data attribute corresponds to a unique Boolean attribute (and vice versa), queries should "respect" this relationship. In other words, if a data attribute is part of a query result schema, then the associated Boolean attribute should also be (and vice versa); and (iii) if some new data attributes are introduced which are in the schema of the query result, also associated new Boolean attribute should be introduced (and vice versa).

The above characterization is simple and provides an easy test (which runs in linear time in the size of the expression) to check whether a query, written as a union of conjunctive queries, is colored. However, such a query can perform arbitrary operations on our representation and ignores the specific semantics of colors and blocks. Our algebra discourages users from expressing such color and block-agnostic queries.

Definition 3. *The colored relational algebra (CRA) consists of the class of colored positive relational algebra queries.*

4.2.2 Color algebra vs Color relational algebra

The color algebra (CA) and the class of color relational algebra (CRA) queries are closely connected. First of all, there exists a translation of any CA query into a CRA query. More specifically,

Theorem 1 (Soundness). For every colored database $\langle \mathbf{D}, \chi \rangle$ and every CA expression Q, there exists a color relational algebra (CRA) expression P such that

$$\mathsf{rep}(Q(\langle \mathbf{D}, \chi \rangle)) = P(\mathsf{rep}(\langle \mathbf{D}, \chi \rangle)).$$

Moreover, given the CA expression Q, the CRA expression P is of polynomial size, to that of Q.

Proof: The proof consists of translating each operator in CA into a CRA query. (see [14] for the translation rules) □.

The previous theorem gives us a way of implementing the CA on top of existing relational DBMS. Our colored DBMS, represents colored databases as described in Section 4 and when a CA query is issued, it translates it first into the corresponding CRA query and then to the equivalent SQL query. Then, the SQL query is executed in a standard relational DBMS.

Our theoretical result is that the CA has the same functionality (expressive power) as the CRA. More specifically:

Theorem 2 (Completeness). For every relational database $\mathbf{D}$ in $\mathcal{S}$, and every color relational algebra expression P, there exists a color algebra expression Q such that

$$\mathsf{rep}^{-1}(P(\mathbf{D})) = Q(\mathsf{rep}^{-1}(\mathbf{D})).$$

Proof: The proof consists of a translation of any CRA query into a CA query and relies heavily on the fact that CRA queries only have a single color attribute in their projection. See [14] for details. □

Without providing a formal proof, we claim that the set of introduced CA operators is minimal. All operators are very natural and hence reducing (if possible) the set of operators would only result in more complex and artificial queries for the user.

5 The MONDRIAN System

In this section, we describe the implementation of the MONDRIAN annotation system. MONDRIAN is implemented on top of the MySQL relational DBMS. The MySQL server is running on

a linux-based Pentium 4 PC (CPU 1.8GHz, 2GB RAM). On top of the relational database, we have implemented a module that accepts as input CA written queries. The module is responsible for translating each such query first to its equivalent CRA query and then to an equivalent SQL query. The resulting SQL query is then sent to the underlying database and is executed against the representation of an annotated database.

Our experiments were conducted using real biological data from the Swissprot [3] database. The relational representation of the Swissprot data was based on the schema of the USCS Genome Browser database [12]. From this relational representation, we extracted two relations for our experiments, namely, relation *Protein* containing 200,000 protein tuples (560MB in size), and relation *Public* that contained four million tuples that concern publications related to proteins (750MB in size). Relation *Protein* has eight attributes in its schema, while relation *Public* has five. We used the above two relations as pools from which we generated three different experimental data sets. Each set contained five unannotated relations for each of the two pools. The sizes of these five relations varied from 10,000 to 50,000 tuples (in 10,000 tuple increments). Thus, the total number of created relations was 30. Each experimental data set allowed for executing experiments with different relation sizes. Different data sets were used to avoid any possible bias in the measurements due to characteristics of the underlying data. Thus, all reported times are averaged over executions involving all three data sets.

A second module of the implementation was responsible for annotating unannotated relations. The annotation process is based on a number of parameters whose values are user specified. Our experience with annotations shows that there are three parameters we should consider while annotating databases. The first parameter, called *MaxNo*, limits the number of blocks that can appear in each annotated tuple. The user specifies a number for *MaxNo* and the system randomly decides, for each tuple, to generate a number of blocks that is less than, or equal to, *MaxNo*. The second parameter, called *AvgNo*, specifies the average size of each generated block. Again, the user specifies a number that is less than, or equal to, the number of attributes in the relation, and the system generates blocks that, in majority, are of size *AvgNo*. The last parameter is the cardinality of $\mathcal{C}$ (the number of colors allowed in the database). In general, $\mathcal{C}$ can vary between one (all blocks have the same color) and the number of blocks in the relation (each block has its own color).

In the above setting, we conducted two sets of experiments.

***(1)*:** The first set of experiments compared the extra cost of executing CA queries over annotated databases versus the cost of executing *equivalent* CRA queries over the corresponding unannotated databases.

***(2)*:** The second set of experiments investigated how the described annotation parameters influence the evaluation cost of CA queries.

All reported times are averaged over five runs of each experiment, over each of the different sets of (un)annotated relation instances (to rule out CPU interference and any bias from using a single instance). For annotated relations, their size is reported as the number of annotated tuples and not the number of representation tuples. The cumulative size of the (un)annotated data sets used in these experiments is 26GB.

5.1 Costs of using colors and blocks

The objective of this experiment is to investigate the added cost we have to pay, in terms of time, due to the annotation of relational databases and the evaluation of color queries (instead of regular relational algebra queries over unannotated databases). We do this by comparing the evaluation cost of operators that are common in the two algebras. The experiment has two parts. Initially, we considered three types of queries, all written in relational algebra, where each type uses one of the operators of selection, projection and join. For example, one query projects on the id and description of a protein, while another selected only proteins of tomatoes. For the join, we used a query that joined *Protein* relation with *Public* relation instances to get proteins along with their associated publications. For each query type, we measured the evaluation time over our experimental data sets.

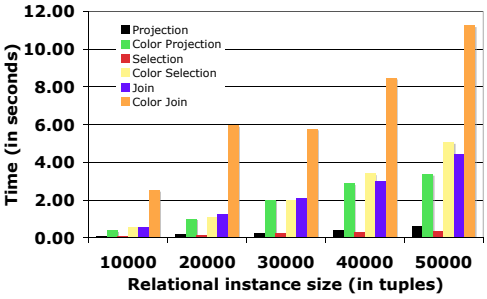


Figure 7: Color vs. normal algebra

In the second part, we annotated the relations used in the first part. While generating the annotated relations, we used what we consider to be representative parameter values. We assumed blocks with *MaxNo* equal to three, since blocks are expected to involve a small number of attributes. We assumed that *AvgNo* is also equal to three. Furthermore, we assumed that $\mathcal{C}$ is 100. Our choice on number of colors is influenced by the fact that different colors often represent different curators. In a scientific database, we don't expect that this number will be greater than 100. Our assumption is validated by the fact that the number of researchers currently curating Swissprot, one of the most heavily curated databases, is close to 40.

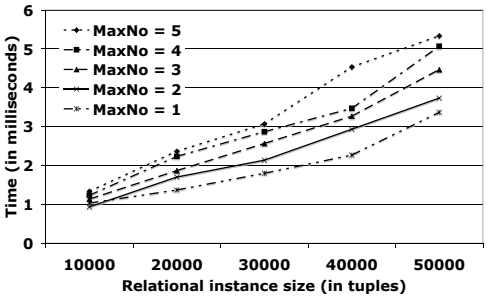
Given the annotated relations, we considered queries that, syntactically, are identical to the queries in the first part of the experiment. However, instead of the normal relational algebra operators, the queries used their color counterparts, i.e., they contained a CA projection instead of a projection, a CA selection instead of a selection and a color join instead of a normal join (note that a color join can be expressed through the select, merge and project CA operators). So, for example, the query that projects on the id and description of proteins was *translated* to a query that projects on the data *and* blocks of these two attributes. Given the queries, we measured their evaluation time over the annotated databases, and we compared these times with the times collected from the first part.

Figure 7 shows the results of this comparison for various relation sizes. Next to the cost of each relational operator, we show the cost of its color counter-part. In general, each color operator costs three times as much as its relational counterpart. There are two main reasons for this behavior. First, remember that color operators are actually applied on the representation of the annotated relation. This representation is bigger in size than the corresponding unannotated relation since it includes one tuple for each color of each block. With *MaxNo* equal to three, annotating a relation with 10,000 tuples results in a relation that is close to 30,000 tuples (assuming single colored blocks). Thus, while a normal projection is applied on 10,000 tuples, a color projection is actually applied on 30,000. Another reason for the extra cost is that color operators must also consider, for each attribute, the corresponding Boolean attributes and operate on them.

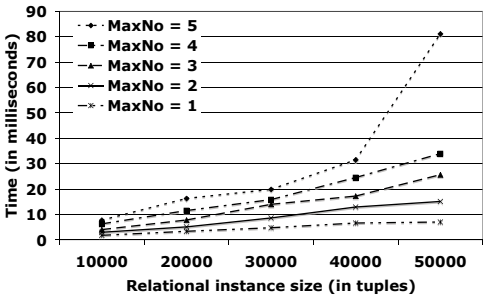
5.2 Query evaluation cost parameters

As the number of blocks and colors vary in an annotated database, we expect that query evaluation will be affected. In what follows, we vary the three annotation parameters and we investigate how exactly these parameters influence color query evaluation.

During these experiments, we considered three different types of color queries. The first type included color queries that involved only the selection operator where the selection was on a data value. Note that the size of the result set of such queries is expected to be independent of blocks. The second type of color queries involved operators that are mostly block-dependent, namely, the operators of block projection and block selection. Finally, the third type of queries involved both block-independent and block-dependent operators. We used three different parameter configurations to annotate relations. For each resulting annotated relation, in each configuration, we executed queries of all three types and measured the corresponding evaluation times. In what



(a) Selection on a data value



(b) Block selection and projection

Figure 8: Varying the maximum number of blocks

follows, we present each parameter configuration and we review our key findings.

***Configuration 1*:** Here, we investigated the influence of the *MaxNo* parameter on query evaluation time. In more detail, we annotated the relations in our experimental data sets once for each value of *MaxNo* between one and five. The *AvgNo* parameter was set to three and the $\mathcal{C}$ was 100. In Figure 8, we show the running times for the first and second type of queries, for the different relation sizes (the third type exhibits the same running time trends as the second type). The main conclusion from these experiments is that the evaluation cost of most CA operators, with the exception of selections on data values, is heavily influenced by the maximum number of blocks per tuple. This is because this number increases the number of tuples in the underlying representation. The trend shown in Figure 8(b) is explained as follows. In the representation, there is one tuple for each color of each block. Assuming single-colored blocks, for an annotated relation with X tuples, there are $(MaxNo + 1) \times X$ representation tuples. As the figure shows, we expect sharp increases in evaluation time, when both *MaxNo* and X increase. In spite of the increase representation size, operations on the data side, like selections on data values, are not influenced by the variance of *MaxNo*, as Figure 8(a) illustrates.

***Configuration 2*:** For our second configuration, we annotated relations of various sizes by varying, this time, the *AvgNo* parameter between the values of one and five. As for the remaining parameters, *MaxNo* was set to three and $\mathcal{C}$ to 100. Again, we executed color queries from all the types and we recorded their evaluation times. Our experiments showed that varying the *AvgNo* parameter had negligible effects on the running times of various queries, for a fixed number of annotated tuples. To a large extent, this is to be expected since any variance of *AvgNo* does not influence the number of tuples in the underlying representation. As *AvgNo* increases, the only change, representation-wise, is that more Boolean attributes are set to 1, instead of 0. It is interesting to note the interaction between the value of *AvgNo* and the size of the result of block projections. As an example, in Figure 9 we show that, for a relation of fixed size, as we increase *AvgNo* we decrease the number of tuples in the result size of a U-type block projection on three attributes. This is because as we increase *AvgNo*, increasingly less blocks are contained within the

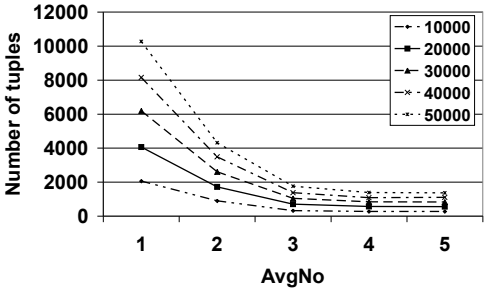


Figure 9: Result size of U-type block projection

projected attribute set.

***Configuration 3*:** Our last configuration considered annotated relations where both *MaxNo* and *AvgNo* were set to three. Here, five different values where considered for $\mathcal{C}$, namely, 1, 10, 100, 1000 and as many colors as there are blocks. Our experiments showed that the parameter has negligible effects on the evaluation times of algebra operators (since again availability of colors does not affect the representation size) with the exception of the block selection operator. As Figure 10(a) shows, when $\mathcal{C}$ is 10, there is a sharp increase on the evaluation time of the operator. The reason for this is shown in Figure 10(b). With less available colors, there is a large number of blocks sharing a color.

6 Conclusions and future work

We have brought the annotating of associations within the realm of relational databases through colored blocks. A query language (color algebra), specifically aimed to query annotated (colored) databases was introduced and its expresiveness was characterized.

The usefulness of the color algebra was illustrated through a variety of examples. The Mondrian annotation management system was conceived and the experiments showed its feasiblity and the effect on the query evaluation of a number of parameters.

This work is only the beginning. We believe that colored databases provide the right framework to answer data provenance questions. Future work will be directed towards showing this. Also, special cases studies obtained by constraining the allowed block structure on a colored database will be performed. Moreover, by attaching a specific semantics to the colored blocks it is most like that MONDRIAN can be applied to a number of settings outside the classical annotation world (e.g., security, access control). Finally, an interesting topic of future work is the extension of our algebra to account for negation.

References

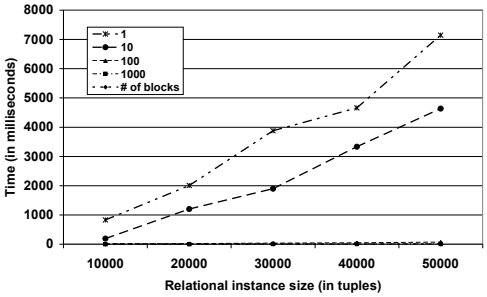
[1] Cathy H. Wu, Lai-Su L. Yeh, et al. The Protein Information Resource. Nucleic Acids Research, 31: 345-347, 2003.

[2] GDB(tm) Human Genome Database [database online]. url http://www.gdb.org/.

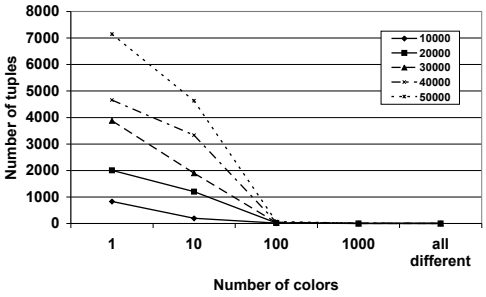
[3] The SWISS-PROT Protein Knowledgebase. URL: http://www.ebi.ac.uk/swissprot/.

[4] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases.* Addison-Wesley, 1995.

[5] Philip A. Bernstein and Thomas Bergstraesser. Meta-data support for data transformations using microsoft repository. *IEEE Data Eng. Bull.*, 22(1):9–14, 1999.



(a) Evaluation time of block selection



(b) Result tuples of block selection

Figure 10: Block selection as $\mathcal{C}$ varies

[6] Deepavali Bhagwat, Laura Chiticariu, Wang-Chew Tan, and Gaurav Vijayvargiya. An Annotation Management System for Relational Databases. In *Proc. of the Int'l Conf. on Very Large Data Bases (VLDB)*, pages 900–911, 2004.

[7] biodas.org. http://biodas.org.

[8] Peter Buneman. The two cultures of digital curation. In *SSDBM*, pages 7–, 2004.

[9] Peter Buneman, Sanjeev Khanna, and Wang Chiew Tan. Why and where: A characterization of data provenance. In *8th International Conference on Database Theory (ICDT)*, pages 316–330, 2001.

[10] Peter Buneman, Sanjeev Khanna, and Wang Chiew Tan. On propagation of deletions and annotations through views. In *Proceedings of the Twenty-first ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pages 150–158, 2002.

[11] Yingwei Cui, Jennifer Widom, and Janet L. Wiener. Tracing the lineage of view data in a warehousing environment. *ACM Trans. Database Syst.*, 25(2):179–227, 2000.

[12] Karolchik D., Baertsch R., Diekhans M., Furey T.S., Hinrichs A., Lu Y.T., Roskin K.M., Schwartz M., Sugnet C.W., Thomas D.J., Weber R.J., Haussler D., and W.J. Kent. The UCSC Genome Browser Database. *Nucl. Acids Res*, 31:51–54, 2003.

[13] Susan Davidson, G. Christian Overton, and Peter Buneman. Challenges in Integrating Biological Data Sources. *Journal of Computational Biology*, 2(4):557–572, 1995.

[14] F. Geerts, A. Kementsietsidis, and D. Milano. MONDRIAN: Annotating and querying databases through colors and blocks. Technical report, School of Informatics, University of Edinburgh, March 2005.

[15] H. V. Jagadish, Laks V. S. Lakshmanan, Monica Scannapieco, Divesh Srivastava, and Nuwee Wiwatwattana. Colorful xml: One hierarchy isn't enough. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 251–262, 2004.

[16] José Kahan, Marja-Riitta Koivunen, Eric Prud'hommeaux, and Ralph R. Swick. Annotea: an open rdf infrastructure for shared web annotations. *Computer Networks*, 39(5):589–608, 2002.

[17] Anastasios Kementsietsidis, Marcelo Arenas, and Renée J. Miller. Data Mapping in Peer-to-Peer Systems: Semantics and Algorithmic Issues. In *Proc. of the Int'l Conf. on the Management of Data (ACM SIGMOD)*, pages 325–336, 2003.

[18] Wang Chiew Tan. Containment of relational queries with annotation propagation. In *9th International Workshop on Database Programming Languages (DBPL)*, pages 37–53, 2003.

[19] Y. R. Wang and S. E. Madnick. A Polygon Model for Heterogeneous Database Systems: the Source Tagging Perspective. In *Proc. of the Int'l Conf. on Very Large Data Bases (VLDB)*, volume 16, Brisbane, Australia, August 1990.

[20] Jennifer Widom. Trio: A system for integrated management of data, accuracy, and lineage. In *Second Biennial Conference on Innovative Data Systems Research (CIDR)*, pages 262–276, 2005.

Appendix

Proof of Theorem 1

We have to prove that for any colored database $\langle \mathbf{D}, \chi \rangle$ and any CRA expression Q, there exists a colored relational algebra expression P such that

$$\mathsf{rep}(Q(\langle \mathbf{D}, \chi \rangle)) = P(\mathsf{rep}(\langle \mathbf{D}, \chi \rangle)).$$

Moreover, given Q, P can be obtained in time polynomial in the size of Q.

We will treat each operator in CRA seperately: If R is a relation name of a colored relation, then we abuse notation and denote by R also the corresponding relation name of its relational representation.

Projection If $Q \equiv \pi_{A_1,\ldots,A_k}(R)$, then $P \equiv \pi_{A_1,\ldots,A_k,B_1,\ldots,B_k,\gamma}(R)$.

Selection If $Q \equiv \sigma_{A_i=a}(R)$, then $P \equiv \sigma_{A_i=a}(R)$. If $Q \equiv \sigma_{A_i=A_j}(R)$, then P consists of the union of four queries, one which selects all blocks not containing A_i and A_j, one which selects all blocks containing A_i and A_j, one which selects all blocks containing A_i but not A_j and for which there exists an equi-colored block (in the same tuple) containing A_j but not A_i, and one which is similar to the previous one but with the roles of A_i and A_j interchanged. More formally,

$$\begin{aligned} P \equiv \sigma_{B_i=0\wedge B_j=0}(R) \cup \sigma_{B_i=1\wedge B_j=1}(R) \cup \\ \pi_{sort(R)}(\sigma_{B_i=1\wedge B_j=0}(R) \bowtie \delta_f(\sigma_{B_i=0\wedge B_j=1}(R)) \cup \\ \pi_{\delta_f(sort(R))}(\sigma_{B_i=1\wedge B_j=0}(R) \bowtie \delta_f(\sigma_{B_i=0\wedge B_j=1}(R)), \end{aligned}$$

where f is a renaming function renaming only the boolean attributes.

Product If $Q \equiv R \times S$, then P is the union of two queries. One which joins the two relations (on the data part) and extends all blocks in an instance $\mathbf{r}$ with zeros in the attributes of an instance $\mathbf{s}$, and a similar one where the roles of $\mathbf{r}$ and $\mathbf{s}$ are interchanged. More formally,

$$P \equiv R \bowtie \pi_{A_1,\ldots,A_k}(\langle \mathbf{s}, \chi_s \rangle) \bowtie (B_1, 0) \bowtie \cdots \bowtie (B_\ell, 0)) \cup S \bowtie (\pi_{A'_1,\ldots,A'_m}(R) \bowtie (B'_1, 0) \bowtie \cdots \bowtie (B'_n, 0)),$$

where A_i (resp A'_i) are the data attributes of S (resp. R), and B_i (resp. B'_i) are the boolean attributes of S (resp. R). Recall that the sorts of R and S are assumed to be disjoint.

Renaming if $Q \equiv \delta_f(R)$, then $P \equiv \delta_f(R)$.

Block projection If $Q \equiv \Pi^L_{A_i}(R)$, then $P \equiv \sigma_{B_i=1}(R)$. Similarly, if $Q \equiv \Pi^U_{A_1,\ldots,A_k}(R)$, then

$$P \equiv \sigma_{\bigwedge_{j=1}^{\ell} B'_j=0}(R),$$

where $\{B'_1, \ldots, B'_\ell\} = sort(R) \setminus \{B_1, \ldots, B_k\}$.

Color selection If $Q \equiv \Sigma_c(R)$, then $P \equiv \bigcup_{A_i} \sigma_{B_i=1\wedge\gamma=c}(R)$, where A_i are the data attributes of R.

Merge If $Q \equiv merge_{Y,Z}(R)$, then

$$P \equiv \pi_{sort(R)\setminus Z\cup f(Z)}\big(\sigma_{\bigwedge_{A_i\in sort(R)\setminus Y} B_i=0}(R) \bowtie \delta_f(\sigma_{\bigwedge_{A_i\in sort(R)\setminus Z} B_i=0}(R))\big),$$

where f is a renaming of the boolean attributes of $sort(R)$.

Recolor If $Q \equiv recolor_R(S)$, then

$$P \equiv \pi_{sort(S)\setminus\{\gamma\}}(S) \bowtie \pi_\gamma(R).$$

Union If $Q \equiv R \cup S$, then $P \equiv R \cup S$.

Since all translations are unions of conjunctive queries and always result in a colored relational database, so are all compositions of these queries. As a result we can easily obtain a translation of an arbitrary CRA query into an equivalent colored relational algebra query on their respective representations. □

Proof of Theorem 2

We have to prove that for every colored relational algebra query P, there exists an expression Q in CRA such that for any colored relational database $\mathbf{D}$, we have that

$$P(\mathbf{D}) = \mathsf{rep}(Q(\mathsf{rep}^{-1}(\mathbf{D}))).$$

It is a well-known fact [4] any expression positive query P can be written as the (finite) union of expressions of the form

$$\pi_{X_1,\ldots,X_k}\sigma_F((Y_1,a_1) \bowtie \cdots \bowtie (Y_\ell,a_\ell) \bowtie \delta_{f_1}(R_1) \bowtie \cdots \bowtie \delta_{f_m}(R_m)), \tag{1}$$

where all $\{Y_1,\ldots,Y_\ell\} \subseteq \{X_1,\ldots,X_k\}$, $Y_i \neq Y_j$ for all $i<j$ and $i,j \in [1,\ell]$. The Y_is do not appear as an attribute in any rewriting $\delta_{f_j}(R_j)$. Moreover, $sort(\delta_{f_i}(R_i)) \cap sort(\delta_{f_j}(R_j)) = \emptyset$. Finally, F is positive selection condition.

So, P is the union of queries P_i of the form (1). We construct the desired CRA expression Q_i for each P_i separately, and obtain the final CRA expression Q by taking the CRA union of all $Q)_i$. Hence, we may assume that P is of the form (1). We obtain the corresponding CRA expression Q as follows:

First, we push as many as possible selections in P through the joins.

- $\sigma_{\delta_{f_i}(R_i.color)=c}$: This can be pushed onto the relation R_i. It is clear that we obtain the equivalent result with the CRA expression $\delta_{g_i}(\Sigma_c(R_i)$. Here, g_i is defined as the restriction of f_i to the data attributes.
- $\sigma_{\delta_{f_i}(R_i.A_j)=a}(R_i)$: This can be pushed onto the relation R_i, We obtain the equivalent result in CRA with the expression $\delta_{g_i}(\sigma_{R_i.A_j=a}(R_i)$.
- $\sigma_{\delta_{f_i}(R_i.B_j)=1}$: This can be pushed onto the relation R_i. It is clear that $\delta_{g_i}(\Pi^L_{(R_i.A_j)}(R_i))$, where A_j is such that $ass(A_i,B_j)$ holds, is the equivalent CRA expression. Similarly, selections on blocks of the form $\sigma_{\delta_{f_i}(R_i.B_j)=0}$ can be simulated by applying the CRA expression $\delta_{g_i}(\Pi^U_{sort(R_i)\setminus R_i.A_j}(R_i)$, where A_j is such that $ass(A_i,B_j)$ holds.

Note that we may assume that these selections do not relate to the constant expression $(Y_1,a_1) \bowtie \cdots \bowtie (Y_\ell,a_\ell)$ since such selections would always yield true or false.

Let S_i, for $i \in [1,m]$, denote the CRA expression consisting of R_i on which the relevant CRA expression corresponding to the selections (as described above) are applied.

Next, we consider the CRA-expression

$$Rel = \delta_{g_1}(S_1) \times \cdots \times \delta_{g_m}(S_m).$$

The constant part $(Y_1,a_1) \bowtie \cdots \bowtie (Y_\ell,a_\ell)$ in P is treated as follows. Here, we distinguish between the following cases: If the color attribute in the projection corresponds to the colored attribute in the constant part, then $(Y_1,a_1) \bowtie \cdots \bowtie (Y_\ell,a_\ell)$ is of the form $(A'_1,a_1) \bowtie \cdots \bowtie (A'_p,a_p) \bowtie (\beta'_1 : b_1) \cdots \bowtie (\beta'_p,b_p) \bowtie (\gamma' : c)$ where $p = (\ell-1)/2)$ (This is because all Y_i's are in the projection). Hence, we need to build a tuple $(a_1,\ldots,a_p)$ with a block (containing the attributes A_i for which $b_i = 1$) of color c. Let $Q_1 = merge_{A_1;A_2}((A_1,a_1,c) \times (A_2,a_2,c))$, and $Q_i = merge_{att(Q_{i-1});A_i}(Q_{i-1} \times (A_i,a_i,c))$. Then, $Const = Q_p$ is the desired CRA expression for the constant part.

In case that the color attribute in the projection does not correspond to the colored attribute, we join $(Y_1,a_1) \bowtie \cdots \bowtie (Y_\ell,a_\ell)$ with (γ,c) and end up with an equivalent expression. We then do the same as in the previous case.

Consider now $Const \times Rel$. We translate the selections in F containing pairwise comparisons into the CRA.

- $\sigma_{\delta_{f_i}(S_i.B_k)=\delta_{f_j}(S_j.B_\ell)}$. To get the equivalent CRA expression, we replace $Const \times Rel$ by $(Const \times Rel_1) \cup (Const \times Rel_2)$, where Rel_1 is obtained by replacing $\delta_{g_i}(S_i) \times \delta_{g_j}(S_j)$ in Rel by

$$\delta_{g_i}(\Pi^L_{A_k}(S_i)) \times \delta_{g_j}(\Pi^L_{A_\ell}(S_j))$$

  and Rel_2 is obtained similarly by replacing $\delta_{g_i}(S_i) \times \delta_{g_j}(S_j)$ in Rel by

$$\delta_{g_i}(\Pi^U_{sort(S_i)\setminus A_k}(S_i)) \times \delta_{g_j}(\Pi^U_{sort(S_j)\setminus A_\ell}(S_j)).$$

  We can push the union outside the projection and deal with the two parts separately. So can ignore the union for the moment and omit the subscript of Rel from now on.

- $\sigma_{\delta_{f_i}(S_i.color)=\delta_{f_j}(S_j.color)}$: Note that when Rel is applied to any colored database instance, the blocks in the result do not cross between attributes of S_i or S_j, To get the blocks of the same color in S_i and S_j we first apply

$$K = merge_{sort(S_i),sort(S_j)}(S_i \times S_j)$$

  which merges all blocks in S_i with blocks in S_j of the same color and removes all other blocks. We then replace S_i (resp. S_j) in Rel by $\pi_{sort(S_i)}(K)$ (resp, $\pi_{sort(S_j)}(K)$). Again, we can push the union outside the expression and can ignore it for the remainder of the proof.

- $\sigma_{\delta_{f_i}(S_i.A_k)=\delta_{f_j}(S_j.A_\ell)}$: To get the corresponding CRA expression we proceed as follows. We first compute

$$K = (\sigma_{S'_i.A_k=(S'_j.A_\ell)}(S'_i \times S'_j) \times recolor_{S_j}(recolor_{S_i}(A, a, c))) \cup (S'_i \times S'_j \times (A, a)).$$

  Hence, in K the tuples satisfying the selection criterium will have an annotated A attribute and will have the original block structure because of the union with the original relations. Next, we consider

$$L = merge_{sort(S_i)\cup sort(S_j),A}(K),$$

  which merges all blocks in $S_i \times S_j$ in the (only those) tuples satisfying the selection criterium with the block in the A attribute. We then extract the desired tuples based on this property with $\Pi^L_A(L)$ and we are done.

Since the projection contains a single color attribute, we know exactly where the colors in the result come from. Suppose that the color attribute of R_1 is taken. As above, we can use the *recolor* operator to recolor all blocks in the other relations (and also in $Const$) with the colors in R_1. As a result every block has at least one common color with blocks in R_1.

We then apply the sequence of merges

$$merge_{A_1\cdots A_p;sort(\delta_{f_1}(R_1))} \circ \cdots \circ merge_{A_1\cdots sort(\delta_{f_{m-1}}(R_{m-1});\delta_{f_m}(R_m)}$$

on the expression we got so far. This creates the right block structure with the right colors.

Finally, we apply the projection corresponding to the projection in the SPJRU-expression and we're done. □