Equilibrium of Redundancy in Relational Model for Optimized Data Retrieval

N. Kojić, D. Milićev
University of Belgrade, School of Electrical Engineering

***Abstract*—Conceptual and relational data models of online transaction processing (OLTP) applications are usually created and maintained following the principle of normalization, which implies avoidance of redundancy. Data retrieval from a disk-based normalized relational database often requires complex and inefficient queries that may cause noticeable performance issues when executed on larger volumes of data. Computer professionals sometimes intentionally trade off the strict normal form to optimize data retrieval queries through error-prone manual tuning and denormalization. We propose a fully automatic optimization approach, based on data redundancy, that relies on a formal cost-benefit model. We prove that finding the optimal level of data redundancy, for given workload statistics, is an NP-Complete optimization problem. A detailed reduction of the problem to binary linear programming is presented in the paper. The proposed optimization approach was evaluated using the TPCE benchmark for OLTP systems. The evaluation has shown that the proposed optimization approach is highly scalable, and that it can be efficiently applied to real-life relational data models.**

***Index Terms* — relational databases, optimization, denormalization, data redundancy, binary linear programming**

I. Introduction

Relational databases are inevitable components of online transaction processing (OLTP) systems and applications. Conceptual models of data-centric applications expressed, for example, in entity-relationship, UML, or another language with similar data modeling semantics, are typically created and maintained following the principle of normalization which implies structuring of models by reducing data redundancy for improved data integrity. Normalization in conceptual models is transferred, either manually or using automated generators, to relational models through direct mapping of classes/entities in conceptual models to the corresponding relations (tables) and attributes (columns) in the relational model.

Data retrieval from a normalized relational schema often targets multiple relations. It leads to either complex queries, in terms of the number of involved relations, or to a larger number of small subsequent queries that implement a gradual navigation over tuples in different relations. Such complex queries or proliferation of small queries, cause performance issues that are especially noticeable on large volumes of data. Researchers and practitioners, especially active in the domain of online analytical processing (OLAP) systems, have been intensively exploring and exploiting denormalization techniques to speed up demanding data retrieval queries against a normalized relational schema [1], [2], [3]. Application of all these and similar techniques in industrial practice usually assumes ad-hoc decisions made by database designers according to their subjective understanding of the (expected) behavior of the application. In our approach, as in some other research approaches described in Section III (but hardly ever in industrial practice, according to our experience), the application of denormalization is based on rigorous engineering methods, which includes profiling of data access operations, detection of (frequent) data access patterns, and formal modeling of the optimization techniques based on the redundancy in the relational model, as one of the denormalization techniques.

The rest of the paper is organized as follows. Section II presents the motivation for this research. A brief overview of existing solutions is given in Section III. Section IV informally describes the problem we are trying to solve. The formal cost-benefit and the optimization model is presented in-detail in section V. The proposed approach was experimentally verified and Section VI gives the overview of the experimental setup and the results. Finally, we draw the conclusions and highlight the most important results in Section VII.

II. Motivation

Strict adherence to normalization may lead to poor performance of an application that persists data in a large relational database. A normalized relational schema is not always suited well to demands served by block-oriented hard drives (magnetic or solid-state drives) [4]. Hard drives, especially magnetic disks, are less efficient for random access over graph-like data structures, which is inevitable when requested data are spread over multiple normalized relations. Though the solid state drives (SSDs) have much better random-access performance than the magnetic disks, the relative usage of the I/O bandwidth remains unchanged [5], [6]. Notwithstanding the fact that the technology of in-memory databases is emerging [7], disk-based database systems are dominant due to affordable price and large storage capacity [8].

Table records in a normalized model usually contain much more columns than are required in particular transactions. This is because the columns in the tables are grouped together because of their conceptual (logical) closeness, without

This research was partially funded by The Ministry of Education, Science and Technological Development of the Republic of Serbia [TR 61416].

considering dynamic data demands at runtime. It can be taken as a presumption that the access to the data (columns) from the same record is significantly cheaper than the access to (related) records in different tables [9], [10], [8]. The described effect may be even more discernible in distributed databases, which often handle data shards (i.e., horizontal partitions of tables) by different nodes (servers), while it is assumed that one record resides entirely on the same node. The response time of data access transactions in distributed databases is significantly affected by the communication overhead, as the cost of crossing the boundary of a node is several orders of magnitude higher than local data access. Consequently, the performance benefit of local data access can be leveraged by keeping copies of required remote data in the local node's database. For example, Google Cloud Spanner [11] explicitly enforces different shards (called "splits" in Spanner) to be intentionally handled by different servers, in order to spread workload and parallelize processing. To compensate for the penalty of such distribution, Spanner supports the notion of interleaved tables: records of a "child" table can be placed just after a record of the related "parent" table to ensure physical locality; one record of the parent table and all its related records of the child table (through the join relationship) are guaranteed to reside in the same split, and thus be handled by the same server. This approach can be treated as a special case of general denormalization by pre-joining relations. It is, however, constrained to embedding entire records of a child table into only one parent table; we aim at a more general approach that allows arbitrary pre-joins of individual attributes (partial records) from many tables, without such limitations.

Optimization of the relational model for a large-scale data-centric application through denormalization cannot be successfully and efficiently accomplished without considering the application data demands [3]. In this paper we focus on data redundancy, as an elementary denormalization technique. With data redundancy, we can optimize (frequent) data retrieval operations at the cost of the increased overhead for (preferably rare) update operations. Finding the optimal level of redundancy becomes a combinatorial optimization problem, which requires an adequate formal modeling [12].

Data-centric applications that use any of the popular object-relational mapping (ORM) frameworks, often suffer from performance issues on large data volumes and for demanding workloads. Popular ORM frameworks map object models and requests straightforwardly to the normalized relational model, which is usually treated as the central artifact, without questioning its normalized schema. Programmers are often forced to put significant effort into performance tuning, aside from the business logic implementation, in order to satisfy (non-functional) performance requirements [13] (e.g. by implementing ad-hoc redundancy, various caching techniques, etc.). Furthermore, programmers often need to bypass the ORM framework to implement complex and time-consuming transactions or queries [14], [15], or to be aware of low-level implementation details and pitfalls of an ORM technology to produce efficient code without so-called anti-patterns [16].

Model-driven software development methodology treats conceptual models as central development artifacts, whilst the source code, as well as the relational model, can be derived (i.e., automatically generated) from it [17]. Such a fully automated model transformation opens new possibilities for organizing the relational model in a more performance-aware way (unusual and unnatural to the engineers, but efficient for the database), whilst the source code, even written with anti-patterns [13], need not necessarily suffer from poor data access performance. It is the responsibility of the ORM layer to intelligently transform object actions to relational queries, relying on redundantly prepared responses to the most frequent and expensive data retrieval operations.

III. Related work

The catalog of various object-relational mapping patterns, presented by Rahayu et al. [18], evaluates their positive and negative effects. Guéhis et al. [19] focused on the data reading anomaly, called the "N+1 problem", that leads to proliferation of small queries when a program generates N+1 separate queries to fetch the data for one object and its N linked objects. They employed profiling of data access patterns, denoted as program summaries, and proposed query rewriting rules that improve query performance. Instead of query rewriting, which requires additional programmer effort, Bernstein et al. presented a data prefetching approach based on data patterns predictors at the ORM level [20]. Their optimization eliminates proliferation of small queries by dynamic data prefetching at runtime. Though this approach decreases the number of database requests, it does not reduce the inherent complexity of accessing multiple relations in the (normalized) relational model. Poor data access performance can be caused by anti-patterns in the source code, as Chen et al. have shown [13]. They identified the following issues: a) the excessive data problem, manifested by fetching superfluous data that is never used, denoted as the mismatch between the required and the retrieved data [21], and b) one-by-one query execution in a loop that results in proliferation of small queries [22]. They also proposed the appropriate code-rewriting techniques.

The ineffectiveness of a normalized relational model is especially noticeable in large-scale databases. Shin and Sanders [1] examined four relational schema denormalization techniques for the optimization of queries in OLAP systems and evaluated conditions of their greatest efficiency: a) collapsing relations, b) partitioning relations, c) adding redundant attributes, and d) adding derived attributes. Their research was inspired by pioneering research on denormalization [23] and various other systematically evaluated denormalization techniques [24], [25], [26]. Analysis of the vertical partitioning in OLAP systems, driven by profiling of transactions, was evaluated by Navathe et al. [27]. All these denormalization techniques are very effective and often applied in the OLAP context, because the update operations almost do not exist in such systems. In OLTP systems, such techniques cannot be applied without considering the increased overhead of the update operations. Dabrowski et al. highlighted importance of data access patterns analysis for relational data modeling in OLTP systems (which they call knowledge-based data

modeling) [3]. They proposed foundations for an expert system for relational modeling driven by workload statistics. While their approach is focused on dividing the starting relational model into clusters of correlated entities, our optimization model is based purely on data redundancy and operates at the attribute level. Boniewicz et al. [2] presented the effectiveness of the data redundancy for the optimization of navigation over deep recursive object hierarchies using redundant links. The approach is effective especially if the hierarchical structure is fixed or updated rarely.

Our optimization model based on data redundancy is intentionally kept independent from a specific implementation of the data redundancy, such as covering indexes, materialized views, redundant columns or redundant tables. All these optimization structures rely on the principle of spatial locality of the correlated data. Selection and combination of these structures has been thoroughly analyzed and covered in several complementary researches. Dash et al. [12] examined the index selection problem and showed that it has a well-structured space of solutions. They described the index advisor – CoPhy that operates at the table level and suggests a set of indices (one per table) to achieve the best response time for a given workload. However, it does not consider join indexes, whilst our approach combines correlated data from multiple relations. Implementation of the data redundancy in a relational model is itself a non-trivial optimization problem of selection between indexes and (materialized) views based on the workload characteristics [28]. Agrawal et al. examined that problem in the context of multidimensional (OLAP) databases and implemented a tuning wizard for SQL Server 2000. Chirkova et al. [29] examined the view selection problem for the conjunctive queries in workloads. The conjunctive queries are typically comprised of chained/joined relations, which we denote as the linear navigation along the chain. The linear navigation is just a special case addressed in our model, as well. Agrawal et al. [30] presented a scalable approach for the optimization of queries based on vertical and horizontal partitioning of database structures driven by the workload characteristics. They analyzed the partial optimization effects of partitioning and indexing, as well as their interaction, and introduced empirical constraints to the cost-benefit model for effective pruning of the solution search space. Advanced partitioning techniques have also been examined in the context of massively distributed computation by Zhou et al. [31]. While their research is focused on partitioning of massive input data, for the sake of distributed processing, our research targets the optimization of data fetching in distributed transactions. Our optimization model is focused on elimination of the communication overhead in distributed environments by exploiting the data redundancy.

IV. Problem Description

In this section we illustrate the problem and the idea of our solution on a simplified example. The running example was extracted and simplified from a real-life web portal for online auctions that we refer to as the application. It is an OLTP system that generates a large amount of data at high transaction rate.

A fragment of the application's conceptual model is shown in UML in Figure 1. Although the model is self-explanatory, we are presenting just the most important conceptual details. When a user (as a *seller*) needs to sell a *product*, he/she can create a new auction and fill in the details of the product being sold. Every product belongs to a *category*, whilst every category can have a superordinate (parent) category (except the root category that does not have any). When the seller completes the new auction, it is further maintained and approved by a *manager*. After the auction is opened, other users (*bidders*) can place their *bids* to it. When the auction closes, the last bid becomes the *winning bid*.

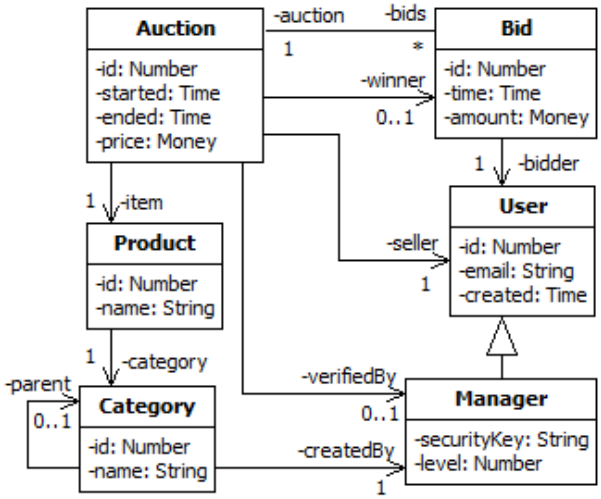

Figure 1 The conceptual model of the online auctions application in UML.

We assume that the application persists data in a relational database whose schema is mapped from the conceptual model such that each class maps to its corresponding table, whilst the associations are implemented, depending on the multiplicity constraints, either using foreign keys or association tables. In Table 1 we show the most important use cases. Each use case has the following details: 1) symbolic name (e.g. *UC1*), 2) short title, 3) frequency (provided by a profiler), and 4) data access section that specifies entity types and their properties.

UC1: Show active auctions (5000)			
Auction:	started, last bid, seller		
Bid:	time, amount, bidder	Product:	name, category
User:	name (bidder or seller)	Category:	name
UC2: Show details of an auction (2000); in 500 cases it was a closed			
Auction:	started, finished, bids, seller, winning	User:	name
Category:	name, parent (full path)	Manager:	name
Bid:	time, amount, bidder	Product:	name, category
UC3: Place a bid to a selected auction (4000)			
Bid:	create (time, amount, auction)		
UC4: Create a new auction (100)			
Product:	create (id, name, category)		
Auction:	create (id, started, product, seller)		
UC5: Close auction (100)			
Bid:	id	Auction:	winner

Table 1 The list of main use cases in the online auction application.

In UC1, the application retrieves active auctions by issuing the query shown in Listing 1. Multiple tables need to be accessed due to the chosen object-relational mapping strategy.

```
select   p.name, a.started, a.ended, us.name, c.name,
         b.time, b.amount, ub.name
from Auction a
join Product p ON (p.id = a.item)
join Category c ON (p.category = c.id)
join User us ON (us.id = a.seller)
join User ub ON (ub.id = b.bidder)
join Bid b ON (b.auction = a.id) [last by b.time]
where  a.opened < now() and a.ended is null
```

Listing 1 Query *Q1*: retrieves summaries of the active auctions, including the last bid for each, as per the specification for UC1.

The use case UC2 is implemented as a server-side operation, rather than a query, because it needs to collect the entire category path (of an arbitrary length) as well as to access the winning bid only if it exists, as shown in Listing 2. Linked objects are fetched by invoking the accessor operations. For instance, the accessor operation `a.item()` loads (navigates to) the product being sold at the auction `a`. It results in a query issued to the database by an object-relational mapping layer.

```
Auction a = findById(1);
Product p = a.item();
Category c = p.category();
while (c != null) {
  collect(c.name()); c = c.parent();
}
User s = a.seller();
collect(p.name(), a.opened(), a.finished(), s.name());
foreach (Bid b in a.bids()) {
  User ub = b.bidder();
  collect(ub.name(), b.amount(), b.time());
}
Bid w = a.winner();
if (w != null) {
  collect(w.amount(), w.time(), w.bidder().name);
}
```

Listing 2 Implementation of UC2 in a Java-like pseudo-language.

From now on, we will treat classes and data types (in a conceptual model) and tables (in a relational model) commonly as entity types. An entity represents an instance of an entity type, with an identity that uniquely distinguishes it from other entities. Objects, data values or table records are entities. Instances of primitive data types are special predefined entities identified by their values (literals). Attributes and associations are treated as *relationships* between *entity types* [32], [33]. Attributes and association ends are jointly referred to as *properties* of entity types. Formally, the properties are allocated as slots in entities. However, we will be using these two terms interchangeably wherever applicable for better and concise readability. Slots store identifiers of linked (referenced) entities. If a property allows maximum one identifier (in a slot), we talk about a functional mapping or a functional relationship. The generalization-specialization relationship can also be modeled as a functional relationship from a subtype entity (e.g. a table record) to a base type entity (e.g. a record in the base table). The identifier of the super type entity is stored in the special (virtual) property h of the subtype entity. Entities (values) of primitive built-in data types are usually embedded in the slots. For instance, a database record embeds a string value instead of pointing to it. Once the record is loaded, all such data type entities become available for the application, without additional data access. However, if an entity is not embedded in a slot, but referenced directly from that slot (e.g., as a symbolic link [3]), the linked entity first needs to be located in the storage (e.g. by determining the location of the database record), and then transferred from the storage.

Once established, functional relationships between entities hold until explicitly changed by the application's write operations. The presence of functional relationships allows packaging of the related entities into (denormalized) database records of predictable size and layout. Having all functionally related entities in the same record may significantly speed up their retrieval and eliminate additional queries, which also improves the percentage of the relevant data retrieved in a single database request [34]. Usually, applications read just subsets of all functionally related entities in individual transactions. Therefore, we introduce the concept of a *data retrieval pattern*, as a model of the data retrieval performed in the navigational sense over the functional properties/slots. The model is aimed to be general enough to cover patterns at different levels of abstraction, as well as to target different data and processing deployments.

In our solution, data retrieval patterns are obtained by profiling a test or a production system in its real or close-to-real (e.g. test, prototype, or pilot) workload environment. They are extracted and analyzed from real execution traces, organized into logical units of processing, that we abstractly refer to as *conversations*. A conversation represents a piece of interaction between the application and the database that takes place in a contiguous time interval, can be executed separately from other such pieces, has a meaningful goal and semantics, and accesses the data in a well-defined and predictable way. It can, for example, represent a batch processing procedure or a transaction, or an entire set of smaller transactions issued from an interactive use case (i.e., a dialog with the user).

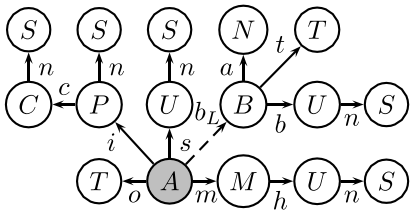

Figure 2 The data retrieval pattern p_1 detected in the traces of UC1. The dashed lines represent the functional relationships derived from the non-functional relationship *bids*: b_L –last bid. The root node is grey.

A data retrieval pattern is represented as a rooted, connected, and directed tree (Figure 2), as it will be formally defined in Section V.A. The vertices represent entities, whilst the directed edges represent properties (slots) of the entities read in a conversation. Every pattern has a root vertex (e.g. A in Figure 2). All the other vertices in the tree are functionally dependent on the root vertex. Each entity in the pattern has its entity type, shown inside its vertex (e.g. A). Similarly, the directed edge s (from A to U) represents a reading of the property/slot s (from an entity of the type A) and retrieving an entity of the type U as the result. For the sake of brevity, we use short identifiers for the entity types and the properties in our sample conceptual

UML model shown in Figure 1. The naming scheme is given as follows: A – Auction, U – User, M – Manager, P – Product, C – Category, B – Bid, T – Time, S – String, N – number. The edges are denoted as follows: n – name, t – time, o – opened, f – finished, i – item, p – parent, c – category, a – amount, h – inherited, m – manager, s – seller, b – bidder or bids.

The data retrieval pattern p_1 in Figure 2 is obtained by parsing the query *Q1* (Listing 1). It is important to notice that the dashed edge b_L represents a functional relationship that does not exist in the conceptual model originally. It is inferred from the non-functional relationship *bids*, assuming that the profiler can be made smart enough to capture the retrieval of a bid under an invariant condition – the latest bid time in this case.

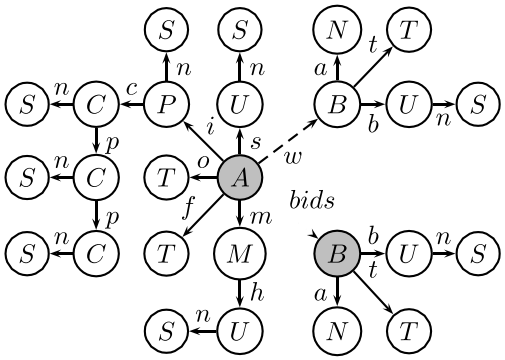

Figure 3 Three data retrieval patterns found in traces for UC2: p_2, p_3, p_4. The root vertices of the patterns are grey.

Figure 3 shows data retrieval patterns captured by profiling execution of the code for UC2 (Listing 2). We can notice three data retrieval patterns that are displayed on the same diagram in Figure 3: p_2 – the small tree rooted at the vertex B in the bottom right corner (determined by $A.bids$ relationship), p_3 – the tree rooted at the vertex A, not including the winning bid's subtree (determined by the edge w), and p_4 – the whole tree rooted at the vertex A (including the winning bid's subtree). The functional relationship w is depicted by the dashed edge just because the winning bid's subtree may not always be present in UC2, depending on whether the auction is closed or not.

Now, let us illustrate effects of denormalization on data retrieval operations by considering one possible denormalized auction data model $\{A^*, B^*, C^*\}$ determined by analyzing the patterns $\{p_1, p_2, p_3, p_4\}$ (see Listing 3). These three denormalized relations can cover the data demands in the workload shown in Table 1. Their columns contain copies of the data from the functionally related relations (in the normalized model).

A^* $(A.s,\ A.s{:}U.n,\ A.i,\ A.m,\ A.m{:}M.h,\ A.m{:}M.h{:}U.n,$
$A.o,\ A.f,\ A.i{:}P.n,\ A.i{:}P.c,\ A.b_L,\ A.b_L{:}B.a,$
$A.b_L{:}B.t,\ A.b_L{:}B.b,\ A.b_L{:}B.b{:}U.n,\ A.w,\ A.w{:}B.a,$
$A.w{:}B.t,\ A.w{:}B.b,\ A.w{:}B.b{:}U.n);$
B^* $(B.t,\ B.a,\ B.b,\ B.b{:}U.n);$
C^* $(C.n,\ C.p,\ C.p{:}C.n,\ C.p{:}C.p,\ C.p{:}C.p{:}C.n);$

Listing 3 One possible denormalization of the auction data model.

Let us further consider the expression $a.winner().name$ from Listing 2 (or shortened $a.w.n$), that fetches the winner's name. In the normalized relational model, this expression requires three separate read operations from the relations A, B, and U. However, the winner's name is redundantly stored in the denormalized relation A^* and becomes available as soon as a record from A^* gets loaded. We use a naming convention for such columns that directly reflects the navigation they optimize. For instance, the result of the navigation $a.w.n$ is stored directly in the column $A.w{:}B.b{:}U.n$ in A^*. The name segment $A.w$ denotes the property w of the entity type A, whilst the colon symbol (:) reflects the type of that property, and a navigation to another relation in the normalized (initial) relational model (e.g. B in $A.w{:}B.n$).

We further assume the following metrics in the database: a) 2000 bids per bidder, b) 10,000 auctions per manager, c) 10 bids per auction, d) 10 subcategories per category, e) 500 auctions per seller. In Table 2 we evaluate the effects of the denormalization on each use case separately. The columns r_N and w_N show the numbers of records that the application accesses in the normalized database per one occurrence of each pattern. Similarly, the columns r_D and w_D show the number of records that the application accesses in the denormalized database per one occurrence of each pattern. The columns R_N, W_N,R_D, W_D show the total numbers of records.

UCx	*freq.*	r_N	w_N	r_D	w_D	R_N	W_N	R_D	W_D
UC1	5,000	10	0	2	0	50,000	0	10,000	0
UC2	500	30	0	12	0	15,000	0	6,000	0
	1,500	28	0	12	0	42,000	0	18,000	0
UC3	4,000	0	1	1	2	0	4,000	4,000	8,000
UC4	100	0	2	3	2	0	200	300	200
UC5	100	1	1	1	1	100	100	100	100
Overall	**64,700**					**107,100**	**4,300**	**38,400**	**8,300**

Table 2 The estimate of the benefit for the proposed schema $\{A^*, B^*, C^*\}$.

The overall estimate is that the proposed denormalized schema $\{A^*, B^*, C^*\}$ can eliminate access to 64,700 (of 111,400) records, given the use cases and the distribution of their frequencies. However, we did not consider a potential overhead imposed by changes of the property $U.n$. A value of n may be a seller's name, a bidder's name or a manager's name, depending on the role that the user plays. If just one name changes per each of the three roles, the application needs to update additional 12,600 records, which decreases the overall benefit to 52,100 records. Though the proposed solution yields this benefit, we still cannot judge about its optimality.

Furthermore, our practical experiments have shown that simple record counting is not always an accurate metric, due to the inherent complexity of relational databases and many of their internal aspects and optimizations. Finally, we proposed the denormalized schema $\{A^*, B^*, C^*\}$ by pure intuition and experience in building and optimizing such systems. However, for more complex models, we need an automated approach that can handle much larger number of entities and their relationships and use a more sophisticated cost-benefit model. We will show that the data redundancy selection problem is an NP-Complete combinatorial optimization problem that can be reduced to binary linear programming and solved efficiently by the standard solvers, which makes this approach applicable to complex real-life problem domains.

V. Formal Model

In this section we introduce the formal model of the data retrieval patterns, formulate the cost-benefit model, and describe the reduction to binary linear programming problem.

A. Definitions

Let E be the set of entity types $\{\mathrm{E}_1, \mathrm{E}_2 \ldots \mathrm{E}_\kappa\}$, each of which is a set of entities extended with $null$ element. Let F be the set of (partial) functions $\{f_1, f_2 \ldots f_n\}$, where $f_k: E_i \mapsto E_j$, $E_i, E_j \in E$. Given the partial function $f: X \mapsto Y$, if $x \in X$ does not participate in the relationship, we extend f with $(x, null)$. In order to allow composition of partial functions, we assume that $f(null) = null$ for every f. The functions abstract properties of entity types. Every function f has the following properties: a) w – *write frequency*, denotes the total number of updates of the mapping (i.e., the number of all writes to the property f of any entity x), b) d – *reverse cardinality*, representing the average number of entities in the function's domain mapped to the same *non-null* entity in the function's codomain, c) μ – *data access cost*, defined as $\mu = \lambda + \tau$, where λ represents the cost of locating the entity in the function's codomain, and τ represents the data transfer cost of the located entity. Multivalued relationships can also be modeled by functional relationships as follows. Through profiling, one may conclude that the application frequently reads only certain (preferably small) subset of k values of a multivalued property f (with cardinality n). Thus the multifunction f can be used for derivation of the following k functions: $f[i_1], f[i_2], \ldots, f[i_k]$, while the rest of $n - k$ values are not treated any further.

Two functions f_p and f_q are *chained*, denoted by $f_p \rightarrowtail f_q$, if the domain of f_q is the codomain of f_p. Having a functional chain $f_{i1} \rightarrowtail f_{i2} \rightarrowtail \cdots \rightarrowtail f_{ik}$, we can specify a function composition $f_x = f_{i1} \circ f_{i2} \circ \ldots \circ f_{ik}$. If the domain of f_{i1} is E_x and the codomain of f_{ik} is E_y, given an entity $e_x \in E_x$, we can determine an entity $e_y \in E_y$ by $e_y = f_{ik}(\ldots f_{i2}(f_{i1}(e_x)))$. However, the function f_x maps e_x to e_y directly by $e_y = f_x(e_x)$ along the chain $f_{i1} \rightarrowtail f_{i2} \rightarrowtail \cdots \rightarrowtail f_{ik}$, and eliminates the subsequent functions invocations. Hence, we refer to it as an *optimization*.

A data retrieval pattern is defined as a directed rooted tree $(V, U, M_V, M_U, r, \omega, \theta)$, where V is a set of vertices, U the set of edges, and:

- M_V is a marking of vertices, i.e. a function (not necessarily an injection) $V \to E$, where E is the set of all entity types,
- M_U is a marking of edges, i.e. a function (not necessarily an injection) $U \to F$, where F is the set of all functions,
- $r \in V$ is the root vertex,
- $\omega \in \mathbb{N}$ denotes the number of occurrences of the pattern in a profile (i.e., its frequency), and
- $\theta \in \mathbb{R}, \theta \in [0,1]$ denotes the normalized cost of the pattern (e.g. a cost obtained from the query optimizer and normalized to 1 considering all the patterns).

Let P be the set of all data retrieval patterns and let O be the set of all optimizations. Any subset $C \subseteq O$ represents a *configuration*. The number of all possible configurations is 2^N, where $N = |O|$. Alternatively, configurations can be represented using the concept of *configuration variable*. A *configuration variable* is defined as a vector of N binary variables $c = [o_1, o_2, \ldots, o_N]$, where each binary variable o_x (also called the optimization variable) represents one optimization from O and determines whether that optimization is applied ($o_x = 1$) or not ($o_x = 0$). If the given optimization $f_x = f_{i1} \circ f_{i2} \circ \ldots \circ f_{ik}$ is applied, then the redundant value of the function f_{ik} (the entity e_y or a reference to it) becomes available each time the entity e_x is loaded by the program $(e_x.f_x)$. This way the total cost of the retrieval along the chain is decreased for the cost μ_{ik} of accessing the value of the function f_{ik}. Let $\rho \in \mathbb{R}$, $\rho \in [0,1]$ denote the ratio of the relevant properties (columns) in an entity (record) for a specific data retrieval (e.g. 2 of 10 columns read, $\rho = 0.2$). In row-oriented relational databases, the projection ratio ρ does not (usually) affect the query plan cost, but we extend the definition of the *data access cost* μ with the corrective factor $1 + L(1-\rho)$, $L \in \mathbb{R}, L \geq 0$ to be able to assign a higher cost to data retrievals of database records with less relevant data (and favor data redundancy in such cases), $\mu = (\lambda + \tau) \cdot \big(1 + L(1-\rho)\big)$.

B. The Cost-Benefit Model

The effects of the data redundancy are quantified by the decreased cost of read operations (*read benefit*) and the increased penalty of write operations (*write penalty*).

The read benefit, denoted by $rb(p, c)$, represents a reduction of the data retrieval cost for a pattern p, due to the applied optimizations in the configuration c. *The total read benefit* $rb(c)$, is defined as a cumulative read benefit for all data retrieval patterns affected by the applied optimizations in c. The operational definition of the *total read benefit* is given in (1), where ω represents the pattern frequency.

$$rb(c) = \sum_{p_k \in P} rb(p_k, c)\, \omega_k \qquad (1)$$

The write penalty, denoted by $wp(f_k, c)$, quantifies the increase of the cost of the update operations that maintain the redundant data imposed by the applied optimizations in the configuration c, each time the function f_k changes. *The total write penalty* $wp(c)$ represents a cumulative write penalty of all the functions participating in the applied optimizations. The operational definition of the total write overhead is given in (2), where w_k represents the write frequency of a function f_k.

$$wp(c) = \sum_{f_k \in F} wp(f_k, c)\, w_k \qquad (2)$$

Let $f_{i1} \rightarrowtail f_{i2} \rightarrowtail \cdots \rightarrowtail f_{im}$ be a functional chain isolated from a simple linear navigation pattern which occurs ω times. The set $O = \{f_{12}, \ldots, f_{1m}, f_{23}, \ldots, f_{m-1m}\}$ contains all the possible optimizations derived from the pattern, represented by the configuration variable $c = [o_{12}, \ldots, o_{1m}, o_{23}, \ldots, o_{m-1m}]$. We now consider the configuration $C = \{f_{12}, f_{13}, \ldots, f_{1m}\}$ of those optimizations that start with the function f_{i1}. Each such optimization is defined as $f_{1k} = f_{i1} \circ f_{i2} \circ \ldots \circ f_{ik}$ and maps the root entity directly to the entity at the end of the chain. The configuration C can be represented by the configuration

variable value $c_{1m} = [1,1,\dots,1,0,\dots,0]$, with $m-1$ leading $1s$. The read benefit of the linear pattern for the configuration c_{1m} is estimated by the equation (3), where: $\mu_{ik} = \mu(f_{ik})$.

$$rb(c_{1m}) = \omega \sum_{k=2}^{k=m} \mu_{ik} \quad (3)$$

Each applied optimization f_{1k} eliminates the retrieval cost μ_{ik}. On the other hand, a change of any function participating in the optimizations in c_{1m}, affects the applied optimizations containing that function in the composition. We treat changes of the functions independently (which models the worst case).

$$wp(c_{1m}) = \sum_{k=1}^{k=m} (wb(f_{ik}) + ra(f_{ik}))\, w_{ik} \quad (4)$$

As shown in (4), the write penalty is comprised of the two components (multiplied by the write frequency w_{ik}): a) *write-backward* penalty $wb(f_{ik})$ and b) *read-ahead* penalty $ra(f_{ik})$. The write-backward component determines the cost of updating all the redundant copies of f_{ik}. The number of the entities holding the redundant copies of f_{ik} (considering the optimization f_{1k}) is determined by the product of the backward cardinalities of the functions that precede f_{ik} in the chain. Once the number of the affected records is determined, it is multiplied by the cost factor μ_{i1}, $wb(f_{ik}) = \mu_{i1} \prod_{j=1}^{j=k-1} d_{ij}$. Likewise, the remaining redundant functions $f_{1k+1}, \dots, f_{1m}$ also became outdated. Their new values need to be retrieved from a new sub-path whose root entity is the new value of f_{ik}. The cost of obtaining their new values is determined by the *read-ahead* penalty that corresponds to the retrieval cost in the new sub-path, $ra(f_{ik}) = \sum_{j=k+1}^{j=m} \mu_{ij}$. These two write penalty components do not necessarily appear together. The cost of changing the first function f_{i1} comes down to the read-ahead penalty, whilst the cost of changing the last function f_{im} comes down to the write-backward component. The total write penalty is given in (5), where m is length of the linear pattern.

$$wp(c_{1m}) = \sum_{k=1}^{k=m} \left(\mu_{i1} \prod_{j=1}^{j=k-1} d_{ij} + \sum_{x=k+1}^{x=m} \mu_{ix} \right) w_{ik} \quad (5)$$

The presented cost-benefit equations are illustrated on the sample linear navigation pattern p_x shown in Figure 4.

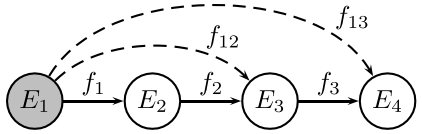

Figure 4 A linear data retrieval pattern p_x for the navigation path $f_1 \rightarrowtail f_2 \rightarrowtail f_3$. The redundant functions are visualized as the dashed edges.

For the considered pattern, the set $O_x = \{f_{12}, f_{13}, f_{23}\}$ contains all the optimizations, whilst $c_x = [o_{12}, o_{13}, o_{23}]$ is the corresponding configuration variable. We consider the fixed configuration $c_{110} = [1,1,0]$ that is visualized by the dashed lines in Figure 4. The optimization f_{23} is not in the configuration ($o_{23} = 0$). For the sake of simplicity, we set the physical parameters as follows: $\tau = 0$, $\lambda = 1$, $\rho = 0.1$ and $L = 0.1$. The concrete cost-benefit equations are shown below.

$rb(c_{110}) = 2 \cdot (1 + 0.09)\omega = 2.18\omega$

$wp(c_{110}) = wp(c_{110}, f_1) + wp(c_{110}, f_2) + wp(c_{110}, f_3) = 2.18w_1 + (1.09 + d_1)w_2 + d_1 d_2 w_3$

where:

$wp(c_{110}, f_1) = w_1(\mu_2 + \mu_3) = 2w_1\lambda(1 + 0.09) = 2.18w_1$

$wp(c_{110}, f_2) = w_2(\mu_2 + \lambda d_1) = (1.09 + d_1)w_2$

$wp(c_{110}, f_3) = \lambda d_1 d_2 w_3 = d_1 d_2 w_3$

Now, let us briefly explain the equations on a concrete chain of three entities: $e_1 \in E_1$, $e_2 \in E_2$ and $e_3 \in E_3$. An update of the property $e_1.f_1$ ("redirection" to a new entity $e_2' \in E_2$) requires reading ahead the new path and collecting the new values of the properties $e_1.f_{12}$ and $e_1.f_{13}$. An update of the property $e_2.f_2$ ("redirection" to a new entity $e_3' \in E_3$) affects the optimizations f_{12} and f_{13}. If f_{12} was the only applied optimization, the read-ahead overhead of the change of the property $e_2.f_2$ would not be required. However, with f_{13} applied, it is necessary to read ahead the new value of $e_1.f_{13}$ from the new entity e_3'. The number of records containing the obsolete values of the properties $e_1.f_{12}$ and $e_1.f_{13}$ is determined by the reverse cardinality of f_1 multiplied by the cost factor μ_1. When the property $e_3.f_3$ changes ("redirection" to a new entity $e_4' \in E_4$), the write penalty has only the write-backward component. The product $d_1 d_2$ determines the number of entities $e_1^* \in E_1$, such that $f_2(f_1(e_1^*)) = e_3$, that have the obsolete values of $e_1^*.f_{13}$.

The impact of the physical parameters in the equations can be eliminated as follows: $\tau = 0$, $\lambda = 1$, $\rho = 1$ and $L = 0$. That way the cost-benefit equations become metrics based on record (request) counting.

$rb(c_{110}) = 2\omega$

$wp(c_{110}) = 2w_1 + (1 + d_1)w_2 + d_1 d_2 w_3$

If $w = 5$ and $d = 2$ for each function, and $\omega = 50$ for the pattern shown in Figure 4, we get the following results: $rb(c_{110}) = 100$ and $wp(c_{110}) = 45$.

Finally, the general cost-benefit metrics based on the record-counting are defined in the equations (6) and (7) respectively.

$$rb(c_{1m}) = (m-1)\omega \quad (6)$$

$$wp(c_{1m}) = \sum_{k=1}^{k=m} \left(\prod_{j=1}^{j=k-1} d_{ij} + (m-k) \right) w_{ik} \quad (7)$$

C. Optimization Model

Now that the cost-benefit model is formally defined, we can formulate the following optimization problem: find a configuration that maximizes the read benefit provided that the write penalty is below the given threshold T. The threshold value defines the lowest requirement for acceptable solutions.

$$MAX \quad rb(c), \quad c = [o_1, o_2, \dots, o_N]$$
$$s.t. \quad rb(c) > wp(c),\ wp(c) \le T,\ T > 0$$

We name our optimization problem *Data Redundancy Equilibrium (DRE)*. To be able to solve DRE problem, we need to incorporate the binary optimization variables into the cost-benefit equations. That way we can quantify performance impact of the optimizations analytically. The read-benefit equation for the pattern in Figure 4 thus becomes:

$rb(c) = \omega \cdot (\mu_1 E_1^{opt} + \mu_2 E_2^{opt} + \mu_3 E_3^{opt})$.

Each binary variable E_k^{opt} determines whether the access to

an entity $e_k \in E_k$ in the pattern is optimized or not; if it is optimized, the retrieval cost is decreased. (These variables have the visual decoration opt in the superscript.) For the given linear pattern, the initial constraint is $E_1^{opt} \stackrel{\text{def}}{=} 0$, because the application needs to load the first entity in the chain at least. Loading of the entity e_2 can be eliminated if all its properties, required in the pattern, are already loaded in the entity e_1, provided that the optimization f_{12} is applied. We formulate this constraint analytically as $E_2^{opt} = o_{12}$. Furthermore, the access to the entity $e_3 \in E_3$ is not required if: a) the optimization f_{13} is applied and the entity e_1 is loaded ($e_1.f_{12}$), or b) the optimization f_{23} is applied and the entity e_2 is loaded ($e_2.f_{23}$). We formulate this constraint analytically as $E_3^{opt} = o_{13} \vee (o_{23} \wedge \neg o_{12})$. Consequently, the optimization f_{23} makes sense only if f_{12} is not applied. The condition $\neg o_{12}$ ensures that the entity e_2 is loaded by the program. If an access to the entity e_2 is not optimized, denoted by $\neg E_2^{opt}$, the access to its property f_2 should not be optimized either, and vice versa. In general, the access to an entity in a pattern is optimized if all its properties (functions), present in the pattern, are optimized. Otherwise, none of the properties should be optimized. For that purpose, we introduce the optimization variables $f_{x1}^{opt}, f_{x2}^{opt}, \dots, f_{xk}^{opt}$ that correspond to the properties $f_{x1}, f_{x2}, \dots, f_{xk}$ of an entity type E_x. The general entity optimization constraint is formulated with the following equations: $E_x^{opt} = f_{x1}^{opt} \wedge f_{x2}^{opt} \wedge \dots \wedge f_{xk}^{opt}$, and $f_{x1}^{opt} \oplus f_{x2}^{opt} \oplus \dots \oplus f_{xk}^{opt} = 0$.

The final transformed cost-benefit equations for the pattern p_x in Figure 4 are shown below ($\tau = 0, \lambda = 1, \rho = 1, L = 0$).

$$rb(c) = (o_{12} + (o_{13} \vee (o_{23} \wedge \neg o_{12}))) \cdot \omega$$

$$wp(c) = (o_{12} + o_{13}) \cdot w_1 + (d_1(o_{12} \vee o_{13}) + o_{13}) \cdot w_2 + (d_1 d_2 o_{13} + d_2 o_{23}) \cdot w_3$$

Values of binary optimization variables and the logical expressions are treated as small integer values $\{0,1\}$ that can be combined in the arithmetic expressions. Since all the independent variables are binary, DRE problem can be reduced to *binary linear programming* (*BLP*). The reduction to BLP is in-detail presented in the following section.

D. Reduction to Binary Linear Programming

Since data retrieval patterns are rarely linear, but most often complex rooted trees, and since there can be many of them, we need a sophisticated algorithm for deriving the equations for the *BLP* model. The input of the algorithm is the set of patterns *P* and the set of functions F. For the sake of brevity and due to the lack of space, we deliberately omit the physical parameters in the listings. The main complexity of the algorithm and the problem itself especially comes from the interference of optimizations that needs to be properly analytically modeled.

Before describing the algorithm (divided into a few procedures), we first give the key assumptions and definitions. We assume that every entity has the property $type$ that provides its type from E, and that every function has the property ent that provides its domain entity. Every optimization is viewed as an ordered tuple $o = (f_{i1}, f_{i2}, \dots, f_{ik})$ of chained functions. The n-th function in the optimization can be referred to using the one-based indexing $o[n]$ (e.g. $o[1]$ accesses the function f_{i1}). The expression $o[last]$ denotes the last function, whilst the expression $o[i..j]$ denotes a sub-chain of the functions at the indexes from i to j, $i < j$. Each pattern p is represented as the tuple $(V, U, M_V, M_U, r, \omega, \theta)$. The components of the tuple can be referred to using the dot (.) operator (e.g., $p.M_V$). The set of the directed edges U is defined as a set of ordered pairs $\{(src, dst) \mid src, dst \in V\}$, where src indicates the starting vertex, and dst indicates the ending vertex, for each directed edge in U. A projection of the relation U, $U[x]$, yields a subset of the ordered pairs that satisfy the condition x. For instance, a subset of the edges, incident on the root vertex in the pattern p is denoted by $p.U[src = p.r]$. We assume that the following basic utility functions are defined on the set of optimizations O:

- $opt(f_x, f_y)$: a set of optimizations having f_x as the first function and f_y as the last function in the chain;
- $via(f_x)$: a set of optimizations with func. f_x in the chain.

1) Derivation of Optimizations

Algorithm D.1 (Listing 4) derives optimizations from the input pattern p. It starts from the edges incident on the root vertex, $p.U[src = p.r]$, and calls Algorithm D.2 (Listing 5) which creates the optimizations. The time complexity of Algorithm D.1 is $O(D \cdot |V|)$, where D is the longest chain in the pattern, and $|V|$ is the number of entities in the pattern.

Algorithm D.1 $opts\_from\_pattern$

Input: p, a data retrieval pattern
Output: O_p, a subset of optimizations

```
1: for edge in p.U[src = p.r] do
2:     O_p ← O_p ∪ opts_from_subtree(p, edge)
3: end for
4: return O_p
```

Listing 4 The algorithm that derives optimizations from the input pattern p.

Algorithm D.2 (Listing 5) finds all functional paths (chains) in the subtree of the pattern p determined by the *edge* input parameter and transforms them to optimizations.

Algorithm D.2 $opts\_from\_subtree$

Input: p, a pattern
Input: $edge$, a start edge
Output: O_p, set of optimizations

```
1: for x in p.U[src = edge.dst] do
2:     o_x ← p.M_U(edge) ∘ p.M_U(x)
3:     O_p ← O_p ∪ {o_x}
4:     for o_s in opts_from_subtree(p, x) do
5:         O_p ← O_p ∪ {o_s, o_x ∘ o_s}
6:     end for
7: end for
8: return O_p
```

Listing 5 The recursive procedure that derives optimizations from the subtree of the pattern p determined by the $edge$ parameter.

Every recursive invocation of the function starts a traversal of the subtree whose root node is the ending node of the *edge* input parameter. The algorithm first determines the incident

edges on the subtree's root node, $p.U[src = edge.dst]$. The edges are mapped to the corresponding functions using the markings $p.M_U$ in line 2. Derived optimizations are stored to the intermediate set O_P (in the scope of the current recursive invocation, line 3). In line 4, the algorithm makes a recursive call for each edge x determined in line 1. Each optimization collected in the recursive call is added to the result set O_p and appended to the composition o_x (line 5).

2) Derivation of Read-Benefit Equations

Algorithm D.3 (Listing 6) generates a read-benefit equation for a pattern p. The algorithm builds the equation gradually using the maps E_o and F_o to keep partial optimization conditions for the entities and the functions in the pattern, respectively. For each entity in the pattern, the algorithm creates an entity optimization variable and keeps it as a key (*eid*) in the map E_o (line 6). Each entity optimization variable is mapped to a set of the function optimization variables that correspond to the properties of that entity. Each such function optimization variable is also stored as a key (*fid*) in the map F_o (line 5) and maps to a set of binary expressions that specify partial optimization conditions for that function within the pattern p. The partial conditions are specified in the abstract form using the angle brackets $\langle ... \rangle$ notation. Optimizations, entities and functions within the angle brackets transform to the corresponding binary optimization variables at runtime. For instance, the following abstract expression $\langle o \wedge o[1].ent \rangle$ may be transformed to the following concrete expression $o_1 \wedge E_1^{opt}$. Note: the algorithm must ensure that each entity and function optimization variable has a unique name in the entire binary linear programming model (considering all the patterns).

Algorithm D.3 $rb\_equation$

Input: p, data access pattern

1: $F_o : \{(fid, fcond : \{\})\}$
2: $E_o : \{(eid, fcond : \{\})\}$
3: $O_p \leftarrow opts\_from\_pattern(p)$
4: **for** o in O_p **do**
5: $F_o[o[last]] \leftarrow \langle o \wedge \neg o[1].ent \rangle$
6: $E_o[o[last].ent] \leftarrow \langle o[last] \rangle$
7: **end for**
8: **for** kv in F_o **do**
9: $out\ \langle kv.fid = \bigvee_{c \in kv.fcond} c \rangle$
10: **end for**
11: **for** kv in E_o **do**
12: $out\ \langle kv.eid = \bigwedge_{c \in kv.fcond} c \rangle$
13: $out\ \langle \bigoplus_{c \in kv.fcond} c = 0 \rangle$
14: **end for**
15: $out\ \langle rb(p) = p.\omega \cdot \sum_{kv \in E_o} kv.eid \rangle$

Listing 6 The algorithm that composes the read-benefit equation and the corresponding constraints for the given pattern p.

The algorithm first calculates the set of optimizations O_p from the given pattern (line 3). The loop in lines 4-7 iterates through the given optimizations and creates partial optimization conditions considering the last function in each optimization. The last function is optimized if the current optimization o is applied and the first entity is not optimized (which ensures that the program will load it and pick up the optimization, line 5). The optimization variable of the last function in the current optimization is added to the set *fcond* of the corresponding entity optimization variable (line 6). The optimization conditions for the functions are assembled in the loop in lines 8-10. A function is optimized if at least one of its conditions is satisfied (line 9). The loop in lines 11-14 completes the entity optimization conditions. An access to an entity is eliminated if all its functions in the pattern are optimized (line 12). Otherwise, none of the functions should be optimized since partial optimizations do not eliminate access to the entity (line 13). Finally, the read benefit equation is completed in line 15 as the sum of the entity optimization variables multiplied by the pattern's frequency. The time complexity of Algorithm D.3 is the same as for Algorithm D.1.

Algorithm D.4 $wp\_equation$

Input: f, updated function
Input: O, the set of all optimizations

1: $RA : \{(eid, (load : \{\}, skip : \{\}))\}$
2: $WB : \{(fchain, (rc, opts : \{\}))\}$
3: **for** o in $O.via(f)$ **do**
4: **for** p in $o.positions(f)$ **do**
5: **if** $WB[o[1..p-1]] = null$ **then**
6: $WB[o[1..p-1]].rc \leftarrow \prod_{i=p-1}^{i=1} o[i].d$
7: **end if**
8: $WB[o[1..p-1]].opts \leftarrow o$
9: **for** $f_s = o[p+1]$ to $o[last]$ **do**
10: $RA[f_s.ent].load \leftarrow o$
11: **for** $f'_s = o[p+1]$ to f_s **do**
12: $RA[f_s.ent].skip \leftarrow O.opt(f'_s, f_s)$
13: **end for**
14: **end for**
15: **end for**
16: **end for**
17: $wp\_eq \leftarrow [\,]$
18: **for** wb in WB **do**
19: $wp\_eq \leftarrow \langle wb.rc \cdot \bigvee_{o \in wb.opts} o \rangle$
20: **end for**
21: **for** ra in RA **do**
22: $wp\_eq \leftarrow \langle (\bigvee_{o \in ra.load} o) \wedge \neg (\bigvee_{o \in ra.skip} o)) \rangle$
23: **end for**
24: $out\ \langle wp(f) = f.w \cdot \sum_{i=1}^{i=wp\_eq.len} wp\_eq[i] \rangle$

Listing 7 The algorithm for composing the write-penalty linear equations with the logical constraints for the input function.

3) Derivation of Write-Penalty Equations

Algorithm D.4 (Listing 7) generates a write penalty equation for a function f. The equation is built gradually using the maps RA and WB for keeping partial expressions for the read-ahead and the write-backward penalty, respectively. The map RA maps an entity (in a read-ahead subtree) to the sets of optimizations $load$ and $skip$. The $load$ set stores all the optimizations that require access to the entity in the read-ahead subtree when the input function changes. However, the access to that entity can be eliminated (due to possible optimizations

in the read-ahead subtree), if any of optimizations in the set $skip$ is applied. Similarly, the map WB maps a functional chain to its reverse cardinality rc and the set of optimizations $opts$ having that chain as the prefix.

The algorithm first determines optimizations affected by a change of the input function (line 3). The optimizations are processed in the loop in lines 3-16. In line 4, the algorithm determines positions of all occurrences of the input function in the current optimization (this way we handle recursive functional chains). If the current composition prefix is not in WB, the algorithm calculates its reverse cardinality and adds a new entry to WB (lines 5-7). The reverse cardinality contributes to the write penalty only if the current optimization is applied. Hence, we add the current optimization to the $opts$ set for the current function composition prefix (line 8). The loop in lines 9-14 iterates through the function composition suffix of the current optimization and adds the current optimization to the $load$ set for each entity in the read-ahead chain (line 10). In the loop in lines 11-13, the algorithm determines all the optimizations that may eliminate the access to the current entity in the read-ahead tree and adds them to the $skip$ set.

Finally, the algorithm assembles the partial expressions into a complete write penalty equation. The write-backward expressions are assembled in the loop in lines 18-20. In line 19 the algorithm creates a partial write-backward penalty expression as a product of the reverse cardinality and the logical condition that evaluates to 1 if at least one optimization in the $wb.opts$ set is applied. The read-ahead expressions are created in the loop in lines 21-23. An entity is accessed in the read-ahead tree if at least one optimization in the $ra.load$ set is applied, and none of the optimizations in $ra.skip$ is applied (line 22). The write-penalty equation is completed as the sum of the partial write penalty expressions in line 24. The time complexity of Algorithm D.4 is $O(|P| \cdot D^3 \cdot |V|)$, where $|P|$ is the number of patterns, D is the longest chain in a pattern, and $|V|$ is the maximum number of entities in a pattern.

E. Proof of NP-Completeness

We showed that $DRE \leq_{poly} BLP$ by providing the detailed reduction that has the maximum polynomial time complexity $O(|P| \cdot D^3 \cdot |V|)$. Our hypothesis is that DRE belongs to NP-Complete class of problems. We will show that the known NP-Complete problem *Binary Knapsack* can be reduced to DRE in polynomial time, $Binary\ Knapsack \leq_{poly} DRE$.

Let us consider an instance X of Binary Knapsack problem as a set of n items, each of which having the value v_i and the weight w_i. The goal of the optimization is to find a subset of the items $S \subseteq X$ such that the following constraints are satisfied (W is the maximum allowed total weight of items in the knapsack):

$MAX \quad V = \sum_{i=1}^{i=n} v_i \cdot x_i$
$s.t. \quad \sum_{i=1}^{i=n} w_i \cdot x_i \leq W$
$s.t. \quad x_i \in \{0,1\}, v_i, w_i \in \mathbb{R}$

On the other hand, let us consider an instance Y of DRE problem comprised of n linear navigation patterns with two functions $A_i \xrightarrow{f_{bi}} B_i \xrightarrow{f_{ci}} C_i$, each of which occurs ω_i times. We eliminate the physical parameters to simplify the equations. From each pattern we derive an optimization $f_{bi} \circ f_{ci}$. Each such optimization, if applied, yields the partial read benefit $rb_i = \omega_i \cdot o_i$, where o_i is the binary optimization variable. Without the loss of generality, let us suppose that the functions f_{ci} are never updated, whilst each f_{bi} function is updated $w(f_{bi}) \equiv w_{bi}$ times. That way the write penalty metric has the read-ahead component only for each pattern. According to the given assumptions and the preconditions, we formulate the corresponding DRE problem as follows (T represents the write penalty threshold, no interference of the optimizations).

$MAX \quad RB = \sum_{i=1}^{i=n} \omega_i \cdot o_i$
$s.t. \quad \sum_{i=1}^{i=n} w_{bi} \cdot o_i \leq T$
$s.t. \quad o_i \in \{0,1\}, \qquad \omega_i, w_{bi} \in \mathbb{R}$

Comparing the two given problems we can easily define a reduction $Q: X \to Y$ that transforms the instance X of Binary Knapsack problem to the instance Y of DRE problem.

$Q = \{(x_i, o_i), (v_i, \omega_i), (w_i, w_{bi}), (W, T)\}, \ 1 \leq i \leq n.$

We can notice that the reduction Q represents a simple renaming transformation which requires polynomial time $O(n)$. Therefore, we can write the following equivalences:

$$\begin{cases} \sum_{i=1}^{i=n} v_i \cdot x_i \overset{Q}{\Leftrightarrow} \sum_{i=1}^{i=n} \omega_i \cdot o_i \\ \sum_{i=1}^{i=n} w_i \cdot x_i \leq W \overset{Q}{\Leftrightarrow} \sum_{i=1}^{i=n} w_{bi} \cdot o_i \leq T \end{cases}$$

Consequently, whenever there is a solution for the DRE problem instance Y, the corresponding solution for the Binary Knapsack problem instance X exists as well. The existence of the algorithm for Binary Knapsack (Listing 8), that calls DRE as its subroutine, proves that DRE problem is NP-Complete.

Algorithm E.1 Reduction of the Bin. Knapsack to DRE

Input: $X_K = \{(v_1, w_1), (v_2, w_2), ..., (v_n, w_n)\}$
Input: $W_K -$maximum allowed weight
Output: $S_K \subseteq X_K, max \sum v_i$ and $\sum w_i \leq W_K$
Linear reduction:
1: $X_K = \{(v_i, w_i)\} \xrightarrow[w_i \to w_{bi}]{v_i \to \omega_i} Y_E = \{(\omega_i, w_{bi})\}$
2: $W_K \to T_E$
DRE invocation:
3: $S_E = DRE(Y_E, T_E)$, where $S_E \subseteq Y_E$
Linear reverse result mapping:
4: $S_E = \{(\omega_i, w_{bi})\} \xrightarrow[w_{bi} \to w_i]{\omega_i \to v_i} S_K = \{(v_i, w_i)\}$
5: **return** S_K

Listing 8 Reduction of the Binary Knapsack problem to DRE problem.

F. The algorithm in action - example

Now that we gave the detailed reduction to binary linear programming, we present a formal procedure for solving the optimization problem illustrated in Section IV. The set of all possible optimizations is derived from the patterns $p1, p2, p3, p4$ by Algorithm D.1 and shown in Listing 9. Algorithm D.3 generated 5 read benefit equations and Algorithm D.4 generated 24 write penalty equations. The space of valid configurations is bounded by 206 constraints. Since the generated BLP program is large, we illustrate just a few the most representative equations.

The read benefit equation for the pattern p_1 is following.

$$rb(p_1, c_x) = 5000 \cdot \left(A_{p1}^{opt} + B_{p1}^{opt} + P_{p1}^{opt} + C_{p1}^{opt} + U_{p1s}^{opt} + U_{p1b}^{opt} + U_{p1m}^{opt} + M_{p1}^{opt}\right)$$

The optimization variable B_{p1}^{opt} for the entity B in the pattern p_1 is following (along with the contextual constraint).

$B_{p1}^{opt} = \left(o_{13} \wedge \neg A_{p1}^{opt}\right) \wedge \left(o_{14} \wedge \neg A_{p1}^{opt}\right) \wedge \left(o_{17} \wedge \neg A_{p1}^{opt}\right)$, and $\left(o_{13} \wedge \neg A_{p1}^{opt}\right) \oplus \left(o_{14} \wedge \neg A_{p1}^{opt}\right) \oplus \left(o_{17} \wedge \neg A_{p1}^{opt}\right) = 0$.

The write penalty for b_L is given below. Since that function is the first in the function composition, the penalty of updates includes only the read-ahead component.

$$wp(b_L, c_x) = 4000 \cdot [(o_{16} \wedge \neg o_7) + (o_{13} \vee o_{14} \vee o_{16} \vee o_{17})]$$

A change of the function b_L requires reading of the new bid's tree. The entity B is read if the condition $o_{13} \vee o_{14} \vee o_{16} \vee o_{17}$ is satisfied. Similarly, the entity U is loaded if o_{16} is applied and the function n is not redundantly stored in the record for the entity B, as indicated in the condition $o_{16} \wedge \neg o_7$.

The following equation illustrates the write penalty for the function c, which has both the write-backward and the read-ahead components.

$$wp(c, c_x) = (o_2 \vee o_4 \vee o_{20} \vee o_{23} \vee o_{26} \vee o_{30}) \cdot d_i + (o_3 \vee o_4 \vee o_{19} \vee o_{20} \vee o_{22} \vee o_{23} \vee o_{25} \vee o_{26} \vee o_{29} \vee o_{30}) + \left((o_{22} \vee o_{23} \vee o_{25} \vee o_{26} \vee o_{29} \vee o_{30}) \wedge (\neg o_{21} \wedge \neg o_{24})\right) + \left((o_{29} \vee o_{30}) \wedge (\neg o_{21} \wedge \neg o_{28})\right).$$

$o_{17} = b_L \circ a$	$o_{20} = i \circ c \circ p$	$o_7 = b \circ n$
$o_{14} = b_L \circ b$	$o_{23} = i \circ c \circ p \circ n$	$o_{21} = p \circ n$
$o_{16} = b_L \circ b \circ n$	$o_{26} = i \circ c \circ p \circ p$	$o_{24} = p \circ p$
$o_{13} = b_L \circ t$	$o_{30} = i \circ c \circ p \circ p \circ n$	$o_{28} = p \circ p \circ n$
$o_{10} = m \circ h$	$o_{29} = c \circ p \circ p \circ n$	$o_{11} = h \circ n$
$o_{12} = m \circ h \circ n$	$o_{33} = w \circ b \circ n$	$o_3 = c \circ n$
$o_1 = i \circ n$	$o_{35} = w \circ t$	$o_{19} = c \circ p$
$o_2 = i \circ c$	$o_{34} = w \circ a$	$o_{22} = c \circ p \circ n$
$o_4 = i \circ c \circ n$	$o_{31} = w \circ b$	$o_{25} = c \circ p \circ p$
		$o_{18} = s \circ n$

Listing 9 The set of all possible optimization derived from the auction data retrieval patterns $p1$, $p2$, $p3$, $p4$.

We used Gurobi solver [35] to solve this binary linear program and it converged to the optimal solution (58,240) in 5 iterations spending around 0.05s. The optimizations in the final solution are shown below and depicted in Figure 5. Intuitively, we can say that if the given workload was replayed against the denormalized relational model, as suggested in the solution, the application would eliminate loading of 58,240 records.

$o_1 = i \circ n$	$o_7 = b \circ n$	$o_{11} = h \circ n$
$o_2 = i \circ c$	$o_{35} = w \circ t$	$o_{17} = b_L \circ a$
$o_{21} = p \circ n$	$o_{34} = w \circ a$	$o_{14} = b_L \circ b$
$o_{24} = p \circ p$	$o_{31} = w \circ b$	$o_{13} = b_L \circ t$
$o_{28} = p \circ p \circ n$		

Listing 10 Optimal denormalization for the auction application model.

	p_1	p_2	p_3	p_4
ω	5,000	30,000	1,500	500
rb	25,000	30,000	7,500	3,000

Table 3 Frequencies and read benefits for the auction patterns obtained by the optimal denormalization shown in Listing 10.

The cumulative write penalty for all the functions was 7,260. Compared to the solution in Section IV, the optimal solution gives higher benefit than the manually chosen one for 6,140 records. We evaluated our optimization approach also using a real benchmark and presented the results in the next section.

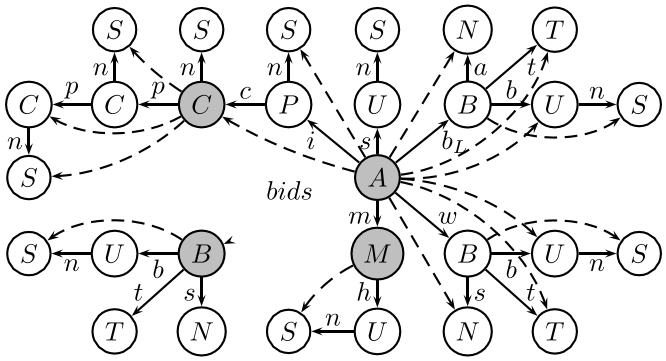

Figure 5 The final optimization solution for the auction model. The dotted lines depict the applied optimizations. The grey nodes depict entities that need to be loaded to satisfy the data demands of the transactions. Note how DRE discarded the optimizations o_{16} and o_{33}, since o_7 is much more frequent.

VI. Experimental Analysis

The proposed optimization methodology was evaluated using the industry-standard TPCE benchmark for OLTP systems (www.tpc.org/tpce). According to the specification [36], the benchmark represents an activity of a stock brokerage firm and is comprised of 12 types of transactions of different complexities, and with frequencies that follow an empirical distribution function. The transactions are: Broker Volume (BV), Customer Position (CP), Market Watch (MW), Security Detail (SD), Trade Lookup (TL), Trade Order (TO), Trade Result (TR), Trade Status (TS), Trade Update (TU), Data Maintenance (DM). They operate on the normalized relational model provided in the specification. The benchmark comes with a data generator and a workload generator. The workload represents a set of the transactions generated according to the empirical frequency distribution function [36].

A. Test Environment Setup

The performance tests were executed against Microsoft SQL Server 2016 on Windows 10. The test machine had 4 Intel i5 CPU cores running at 3.2GHz. The server hosted two databases: (1) a standard (referent) TPCE database with the normalized schema, and (2) a clone of the first database, but denormalized using our optimization approach. Both databases contained identical data, except that the second one additionally contained redundant data created based on the suggested optimizations. Both databases were deployed on two disks: an SSD for the data files and an HDD for the transaction log files. The performance characteristics of the disks are shown in Table 4.

	Seq. RD	Seq. WR	Rand. RD	Rand. WR
SSD	200MB/s	126MB/s	133MB/s	90MB/s
HDD	180MB/s	180MB/s	0.7MB/s	1.7MB/s

Table 4 The throughput values for the SSD and the HDD used in the test.

B. Test Input

The size of the normalized database was 120GB. It was prepared by ingesting around 75GB of raw input data from CSV files created by the data generator. The data generator is

configured to produce the following numbers of records in the referent relations shown below.

CUSTOMER:	20,000	**CUSTOMER_ACCOUNT**:	100,000
TRADE:	172,812,504	**SETTLEMENT**:	172,800,000
COMPANY:	10,000	**SECURITY**:	13,700

The transactions, implemented as batches of SQL statements, are divided into smaller logical pieces called *frames* in the TPCE specification [36]. We implemented each frame as one stored procedure. The transaction mix therefore represented a batch of the stored procedures calls (transactions).

C. Data Retrieval Patterns

Data retrieval patterns were extracted by profiling a control transaction mix of 10,000 transactions using SQL Server Profiler. The number of the transactions in the control mix ensured that each query in the stored procedures appeared at least once in the profile, while keeping the profile still manageable and representative for the analysis. In Table 5 we show 13 data retrieval patterns detected in the control TPCE workload. For each pattern (query) we give its code name (for further referencing in the text), a complexity indicator of the pattern tree, as the number of vertices and the maximum tree depth, a frequency in the control mix, the ratio ρ of relevant data in the pattern, a physical cost obtained from the query plan optimizer (*cost*), and the normalized cost of the pattern (θ) relative to all the patterns in the control mix. The code name is comprised of the three segments: the two-letter transaction name acronym, a frame identifier and a pattern identifier (e.g., the code name BVF1P1 refers to the pattern P1 extracted in the Frame 1 of the *Broker-Volume* transaction).

We also calculated values of the physical parameters ($\tau = 0$, $L = 0.1$, $\lambda = 0.01$), but skip the detailed calculations due to the lack of space. The average record locating cost λ was calculated as a weighted sum of the costs in Table 5. Write frequencies of the properties were obtained by querying the TPCE profile.

			Query parameters			Prediction	
Pattern	**Vertices /Depth**	**Freq.**	ρ	$cost$	θ	**Records**	**Cost**
BVF1P1	9/5	548	0.34	1.600	0.551	1,096	252.85
CPF1P2	7/3	1,462	0.73	0.170	0.059	1,462	63.20
CPF2P3	9/2	744	0.45	0.088	0.030	0	0.00
MWF1P4	3/2	1215	1.00	0.007	0.002	1,215	1.58
MWF1P5	5/3	117	0.32	0.030	0.010	234	0.85
MWF1P6	3/2	118,274	0.19	0.006	0.002	118,274	154.07
SDF1P7	39/4	1,567	0.73	0.023	0.008	4,701	8.16
SDF1P8	4/2	1,567	0.47	0.014	0.005	3,134	5.74
SDF1P9	3/2	1,567	0.86	0.085	0.003	1,567	3.06
TLF1P10	10/2	1,024	0.47	0.700	0.241	0	0.00
TRF3P11	3/2	428	1.00	0.200	0.002	428	0.55
TSF1P12	13/3	2,130	0.43	0.240	0.083	0	0.00
TSF1P13	6/2	2,130	0.23	0.009	0.003	4,260	4.68

Table 5 Data retrieval patterns detected in the test TPCE workload: P1 (p.89), P2 (p.87), P3 (p.89), P4 (p.99), P5 (p.99), P6 (p.99). P7 (p.105), P8 (p.106), P9 (p.108), P10 (p.112), P11 (p.154), P12 (p.162), P13 (p.163). For each pattern we provide a page number in the TPCE specification [36].

Given the patterns, the frequencies of the patterns, the write frequencies of the functions, and the physical parameters, our optimization engine derived 198 optimizations and created a BLP program with 598 constraints. The Gurobi solver found the optimal solution in 0.07s (running on an ordinary PC) by selecting 63 of the 198 determined optimizations. The following patterns (10 of 13) were affected by the suggested optimizations: B1F1P1, CPF1P2, MWF1P4, MWF1P5, MWF1P6, SDF1P7, SDF1P9, SDF1P9, TRF3P11 and TSF1P13. The BLP program produced two benefit predictions of the optimizations: one based on the record counting and the other one, more sophisticated, that considered the physical parameters, as shown in the last two columns in Table 5. For the found configuration, our optimization engine created a denormalization (SQL) script for the second test database. The affected queries and updates in the TPCE transactions were adjusted to exploit the implemented optimizations.

D. Performance Tests

For the performance testing, we generated a transaction mix (workload) of 100,000 transactions. In order to highlight the effects of the optimizations on each particular transaction type, we also divided the transaction mix into the partial homogenous batches shown in Table 6, such that each batch contained only one type of the transactions.

Each partial batch was run several times on the normalized database and several times on the (optimally) denormalized database. The database server was restarted each time before switching the database. The following performance parameters were measured for each batch (using SQL Server Profiler): the average number of logical reads per transaction (*Rd*), the average number of logical writes per transaction (*Wr*) and the batch response time (*T*). Logical reads and writes correspond to the number of (logical) database page accesses performed by the DBMS. They may not necessarily reflect the real I/O traffic, because the pages usually reside in the main memory or in the CPU's cache and can be reused by multiple queries. They rather depict the implicit complexity of the queries.

		Normalized			Optimal denorm.			Fully denorm.		
Tr.	**Freq.**	**Rd**	**Wr**	**T_N[s]**	**Rd**	**Wr**	**T_O[s]**	**Rd**	**Wr**	**T_F[s]**
BV	4,946	47,718	0	1,762	25,061	0	193	62,439	0	228
CP	13,116	226	0	27	212	0	25	551	0	54
MW	18,165	1,422	0	64	649	0	48	648	0	48
SD	14,128	2,214	0	244	2,137	0	226	2,126	0	230
TL	8,069	405	0	159	422	0	153	628	0	210
TO	10,191	100	5	18	104	6	19	112	6	20
TR	10,192	155	5	53	156	5	56	127	6	99
TS	19,175	533	0	79	431	0	73	166	0	137
TU	2,018	1,089	18	61	1,087	18	60	1,111	31	106
DM	1,000	14,757	4	65	14,609	4.4	69	14,307	4.6	71

Table 6 TPCE performance results showed for the normalized, the optimally denormalized and the fully denormalized relational TPCE database.

In Table 6 we can notice that the highest response time improvement of the benchmark executed on the optimally denormalized TPCE database is achieved for the batches that were fully optimized by packaging all the patterns into single database records (BV – 89%, MW – 25%). In the SD (7%) and TS (8%) batches the data retrieval was not fully optimized, and JOINs were still needed. The performance results for the batches *TO*, *TR* and *DM*, aimed for updates, were degraded (but just for 6%) due to the increased data maintenance overhead.

We also tested the TPCE workload on a fully denormalized database, implemented by packaging the entire patterns into single database records, to check whether the optimal denormalization truly leads to the best performance. Therefore, we additionally applied all those optimizations that were discarded in the optimal solution. The discarded optimizations mainly had the root entities in the huge tables, such as TRADE and TRADE_HISTORY. After adding the redundant data (from

much smaller tables) to these big tables, the database size significantly increased, as shown in Table 7. The fully denormalized relation TRADE occupied 162GB, whilst the fully denormalized relation TRADE_HISTORY occupied 64GB. Querying from these big tables was impossible without appropriate covering indexes. The biggest covering index on TRADE was ~50GB large, whilst the biggest covering index on TRADE_HISTORY was ~26GB. Such large indexes induced the higher I/O traffic during the performance test. The performance results of the TPCE workload on the fully denormalized database are shown in Table 6.

Finally, the full workload with 100,000 mixed transactions was executed on all the three databases and the results are given in Table 7. The response time of the workload was improved by 65% on the optimally denormalized database, and by 57% on the fully denormalized database. The optimally denormalized database was 25% larger, whilst the fully denormalized database was 316% larger than the normalized database. Although the optimal denormalization gave just 19% better response time than the greedy denormalization (because the redundant data in the largest tables never got updated), the optimal denormalization was much more effective in controlling the database space expansion.

TPCE schema	T [s]	ΔT [%]	Size [GB]	ΔSize [%]
normalized	2,910	--	120	--
optimally denormalized	1,005	-65	150	+25
fully denormalized	1,240	-57	380	+316

Table 7 The impact of a denormalization strategy on the TPCE benchmark response time and the TPCE database size. The percentages quantify changes of the response time and the database size relative to the normalized model.

E. Analysis of the Cost-Benefit Model

We also analyzed the quality of the prediction, provided by the cost-benefit model, on the test sample of 10,000 transactions. The results are shown in Table 8. The cumulative logical reads are shown in column *N* for the normalized database, and in column *D* for the denormalized database.

		Measured records				Prediction	
Tr.	**Freq.**	**N [pages]**	**D [pages]**	**ΔP=N-D**	**ΔR=ΔP/86**	**Cost**	**Records**
BV	495	23,620,749	12,405,379	11,215,369	130,411	252.85	1,096
CP	1312	297,332	278,887	18,445	214	63.20	1,462
MW	1817	1,754,453	349,778	1,404,675	16,333	156.50	120,500
SD	1413	161,897	49,870	112,027	1,303	16.96	9,300
TR	1019	42,058	30,645	11,413	133	0.55	428
TS	1918	1,023,405	828,559	194,846	2,266	4.68	4,260
Overall		**26,899,894**	**13,943,118**	**12,956,775**	**150,660**		

Table 8 Summary of the measured logical reads and writes in the transaction mix of 10,000 transactions (N – normalized, D – denormalized).

The cumulative decrease of the logical reads in Table 8 is equal to 12,956,776 (pages). In order to compare this result with our prediction based on record-counting, we analyzed the average number of records per page for each table that appeared in the patterns and calculated ~86 records per page on average. When the logical counters are divided by the given coefficient, it yields the benefit of 150,660 records. On the other hand, our record-counting prediction calculated the optimal objective of 122,279 records (our metrics did not consider index pages).

We checked the correlation between the predicted and the measured values using the Pearson's correlation function (Table 9). The low correlation between the record counting prediction and the measured parameters clearly confirms that record counting is not sufficient as the only metric. However, it does not mean that this metric is not useful in other contexts, where record or request counting makes more sense, which will be the subject of our further research. On the other hand, the record counting cost-benefit model, enhanced by the physical parameters, led to the strong correlation of the predicted values with the measured values, as shown in Table 9. Consequently, this analysis confirms that the proposed enhanced cost-benefit model gives meaningful and real estimates.

		Measured parameters		
		$\Delta T=T_N-T_O$	$d_T=\Delta T/T_N$	ΔR
Prediction	**Cost**	0.83	0.90	0.87
	Records	-0.21	0.02	-0.10

Table 9 Pearson's correlation of the predicted benefit and the measured performance parameters shown in Table 6.

VII. Conclusion

In this paper we presented a methodology for optimization of data retrieval from relational databases in OLTP systems based on the data redundancy denormalization technique. Selection of the optimizations that trade off the increased overhead of rare update operations for better performance of more frequent read operations is driven by the proposed formal cost-benefit model. We proved that finding the optimal level of redundancy in the relational model represents an NP-complete optimization problem and gave the detailed reduction to binary linear programming problem. The proposed optimization methodology was experimentally evaluated using the TPCE benchmark (on SQL Server 2016). The response time of the benchmark was improved by 65%, whilst the prediction of cost-benefit model was strongly correlated with the measured parameters according to the maximum Pearson's correlation coefficient 0.9. We have also shown that the optimal denormalization is superior relative to the full denormalization in terms of both the response time and the space consumption. The experimental evaluation has confirmed that the proposed optimization approach is highly scalable, and that it can be efficiently applied to real-life relational data models.

VIII. References

[1] S. K. Shin and G. L. Sanders, "Denormalization strategies for data retrieval from data warehouses.," *Decision Support Systems*, vol. 42, no. 1, pp. 267-282, 2006.

[2] A. Boniewicz, P. Wisniewski and K. Stencel, "On redundant data for faster recursive querying via ORM systems," in *Computer Science and Information Systems (FedCSIS)*, Kraków, 2013.

[3] C. E. Dabrowski, D. K. Jefferson, J. V. Carlis and S. T. March, "Integrating a knowledge-based component into a physical database design system," *Information & Management*, vol. 17, no. 2, pp. 71-86, 1989.

[4] S. Agrawal, S. Chaudhuri, A. Das and V. Narasayya, "Automating layout of relational databases," in *Proceedings of the 19th IEEE International Conference on Data Engineering*, 2003.

[5] D. Tsirogiannis, S. Harizopoulos, M. A. Shah, J. L. Wiener and G. Graefe, "Query processing techniques for solid state drives," in *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, 2009.

[6] J. Do, Y.-S. Kee, J. M. Patel, C. Park, K. Park and D. J. DeWitt, "Query processing on smart ssds: Opportunities and challenges," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, 2013.

[7] H. Garcia-Molina and K. Salem, "Main memory database systems: An overview," *IEEE Transactions on knowledge and data engineering*, vol. 4, no. 6, pp. 509-516, 1992.

[8] H. Plattner, "A common database approach for OLTP and OLAP using an in-memory column database," in *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, 2009.

[9] D. J. Abadi, P. A. Boncz and S. Harizopoulos, "Column-oriented database systems," in *Proceedings of the VLDB Endowment 2*, 2009.

[10] S. Harizopoulos, V. Liang, D. J. Abadi and S. Madden, "Performance tradeoffs in read-optimized databases," in *Proceedings of the 32nd international conference on Very large databases*, 2006.

[11] https://cloud.google.com/spanner/docs/schema-and-data-model. [Online]. Available: https://cloud.google.com/spanner/docs/schema-and-data-model.

[12] D. Dash, N. Polyzotis and A. Ailamaki, "CoPhy: a scalable, portable, and interactive index advisor for large workloads," *Proceedings of the VLDB Endowment 4*, vol. 4, no. 6, pp. 362-372, 2011.

[13] T.-H. Chen, W. Shang, Z. M. Jiang, A. E. Hassan, M. Nasser and P. Flora, "Detecting performance anti-patterns for applications developed using object-relational mapping," in *Proceedings of the 36th International Conference on Software Engineering*, Hyderabad, 2014.

[14] B. Baesens, A. Backiel and S. v. Broucke, "The state of database access in Java: Passchendaele revisited," Cutter Consortium, Research Center for Management Informatics (LIRIS), Leuven, 2015.

[15] D. Vohra, B. Baesens, A. Backiel and S. v. Broucke, Beginning Java programming: the object-oriented approach, Wiley & Sons, 2015.

[16] P. Wegrzynowicz, "Performance antipatterns of one to many association in hibernate," in *Federated Conference on Computer Science and Information Systems (FedCSIS)*, 2013.

[17] D. Milicev, Model-driven development with executable UML, John Wiley & Sons, 2009.

[18] J. W. Rahayu, E. Chang, T. S. Dillon and D. Taniar, "Performance evaluation of the object-relational transformation methodology," *Data & Knowledge Engineering*, vol. 38, no. 3, pp. 265-300, 2001.

[19] S. Guéhis, V. Goasdoué-Thion and a. P. Rigaux, "Speeding-up data-driven applications with program summaries," in *Proceedings of the 2009 International Database Engineering & Applications Symposium ACM*, Calabria, 2009.

[20] P. A. Bernstein, S. Pal and D. Shutt, "Context-based prefetch–an optimization for implementing objects on relations," *The VLDB Journal - The International Journal on Very Large Databases*, vol. 9, no. 3, pp. 177-189, 2000.

[21] T.-H. Chen, W. Shang, Z. M. Jiang, A. E. Hassan, M. Nasser and P. Flora, "Finding and evaluating the performance impact of redundant data access for applications that are developed using object-relational mapping frameworks," *IEEE Transactions on Software Engineering*, vol. 42, no. 12, pp. 1148-1161, 2016.

[22] T.-H. Chen, "Improving the quality of large-scale database-centric software systems by analyzing database access code," in *Data Engineering Workshops (ICDEW), 2015 31st IEEE International Conference*, 2015.

[23] M. Schkolnick and P. Sorenson, "Denormalization: a performance-oriented database design technique," in *Proceedings of the AICA*, Bologna, Italy, 1980.

[24] M. Hanus, "To normalize or denormalize, that is the question," in *Proceedings of 19th International Conference for the Management and Performance Evaluation of Enterprise Computing Systems*, San Diego, CA, 1994.

[25] U. Rodgers, "Denormalization: why, what, and how?," *Database Programming & Design*, vol. 12, p. 46–53, 1989.

[26] S. Agarwal, C. Keene and K. M. Arthur, "Architecting object applications for high performance with relational databases," in *OOPSLA Workshop on Object Database Behaviour, Benchmarks, and Performance*, Austin, 1995.

[27] S. Navathe, S. Ceri, G. Wiederhold and J. Dou, "Vertical partitioning algorithms for database design," *ACM Transactions on Database Systems (TODS)*, vol. 4, pp. 680-710, 1984.

[28] S. Agarawal, S. Chaudhuri and V. Narasayya, "Automated selection of materialized views and indexes for SQL databses," in *Proceedings of 26th International Conference on Very Large Databases*, Cairo, Egypt, 2000.

[29] R. Chirkova, A. Y. Halevy and S. Dan, "A formal perspective on the view selection problem," in *VLDB*, 2001.

[30] S. Agrawal, N. Vivek and Y. Beverly, "Integrating vertical and horizontal partitioning into automated physical database design," in *Proceedings of the 2004 ACM SIGMOD international conference on Management of Data*, 2014.

[31] J. Zhou, B. Nicolas and L. Wei, "Advanced partitioning techniques for massively distributed computation," in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, 2012.

[32] A. Olivé, Conceptual Modeling of Information Systems, New York, Inc., Secaucus, NJ, USA: Springer-Verlag, 2007.

[33] O. M. Group, "Unified Modeling Language," OMG, 2017.

[34] D. J. Abadi, D. S. Myers, D. J. DeWitt and S. R. Madden, "Materialization strategies in a column-oriented DBMS," in *ICDE 2007. IEEE 23rd International Conference on Data Engineering*, 2007.

[35] "Gurobi Solver," Guroby Optimization, [Online]. Available: http://www.gurobi.com/.

[36] TPC, "TPCE," 2010. [Online]. Available: http://www.tpc.org/tpc_documents_current_versions/pdf/tpc-e_v1.14.0.pdf.

Nemanja M. Kojić is a PhD student at the University of Belgrade. He received his B.S. degree in 2008, and M.S. degree in 2010, all from the University of Belgrade. He is specialized in system analysis and design of enterprise software systems, reverse engineering and architecting large-scale data-processing solutions for digital forensic and e-discovery.

From 2010 to 2018, he worked as a Teaching Assistant at the University of Belgrade. His research interests are model-based software engineering, data engineering, and optimization problems and applications.

Dragan S. Milićev is Professor at the University of Belgrade, Faculty of Electrical Engineering. He received his dipl. ing. degree in 1993, M.S. in 1995, and PhD in 2001, all from the University of Belgrade. He is specialized in software engineering, model-based engineering, model-driven development, UML, software architecture and design, information systems, and real-time systems. He is a member of the Editorial Board of Springer's Software and System Modeling journal (SoSyM).

He authored three books on object-oriented programming and UML, published in Serbian, and a book in English, published by Wiley/Wrox, entitled "Model-Driven Development with Executable UML".

With more than 30 years of extensive industrial experience in building complex commercial software systems, he has been serving as the chief software architect, project manager, or consultant in a few dozen international projects.