VLDB Journal, 3, 289-323 (1994), Tadao Ichikawa, Editor

© VLDB

Efficiently Instantiating View-Objects From Remote Relational Databases

Byung Suk Lee and Gio Wiederhold

Received June 13, 1991; revised version received, June 13, 1992; accepted June 17, 1993.

Abstract. View-objects are complex objects that are instantiated by delivering a query to a database and converting the query result into a nested structure. In relational databases, query results are conventionally retrieved as a single flat relation, which contains duplicate subtuples in its composite tuples. These duplicate subtuples increase the amount of data to be handled and thus degrade performance. In this article, we describe two new methods that retrieve a query result in structures other than a single flat relation. One method retrieves a set of relation fragments, and the other retrieves a single-nested relation. We first describe their algorithms and cost models, and then present the cost comparison results in a client-server architecture with a relational main memory database residing on a server.

Key Words. Complex object, nested relation, relation fragments, query optimization, client server.

1. Introduction

Relational databases are not sufficient to support non-traditional applications such as engineering or office information systems. In these applications, users often want to deal with information in a form more abstract than relations. An object, taking the form of a user-defined aggregate data structure, is used to program languages as an abstraction mechanism. Wiederhold (1986) noted that views provide a similar abstraction in databases, and proposed to use a *view-object* as an "architectural tool" for interfacing object-based programs and relational databases. Lee and Wiederhold subsequently developed a system model (Lee, 1990; Lee and Wiederhold, 1994) for embodying the view-object concepts. In the system model, a view is not just

Byung Suk Lee, Ph.D., is Assistant Professor, Graduate Programs in Software, University of St. Thomas, St. Paul, MN 55105-1096 and Gio Wiederhold, Ph.D., is Professor, Computer Science Department, Stanford University, Stanford, CA 94305.

a relational query, but it also contains a function—called the *attribute mapping function*—for mapping between object attributes and relation attributes. The query is used to materialize necessary data into a relation, and the function is used to restructure the materialized relation into a nested relation (Abiteboul and Bidoit, 1984; Roth et al., 1988).

The view-object approach provides an effective mechanism for building complex object-based applications on top of relational databases. Applications are built using complex objects as structural units (Haskin and Lorie, 1982; Lorie and Plouffe, 1983; Dittrich and Lorie, 1985; Wilkes et al., 1989; Barsalou and Wiederhold, 1990) and benefit from the nonredundant storage of information in a nested structure (compared with a flat non-nested structure). At the same time, relational databases provide sharing and flexibility, the benefit from which increases as the size of databases increases. Currently there are engineering design applications (Law et al., 1990*a*, 1990*b*, 1991; Singh, 1990) and medical applications (Wiederhold et al., 1990) that are being built at Stanford University as part of the PENGUIN project (Barsalou, 1990). Complex object-based applications run on a client workstation and cache view-objects run from a relational database residing on a server.

There are three problems in the view-object architecture: (1) view update ambiguity, (2) tuple loss, and (3) performance. Sometimes we update cached view-objects and cannot update the underlying relations accordingly, because we have lost information about normalized relation schema while performing joins for view materialization. (Barsalou, 1990; Barsalou et al., 1991). Tuple losses occur for dangling tuples in a view materialization. Frequently the semantics of view-objects require that even a dangling tuple should be retrieved as the result of joins. The authors introduced a left outer join and developed a mechanism for preventing tuple losses (Lee, 1990; Lee and Wiederhold, 1994). The last, but not the least, problem is the performance of view-object caching in the client-server architecture, especially when the network communication overhead is significant. We address the performance problem in this article.

Performance is influenced by three factors: (1) *query processing* on a server, (2) *transmission* of the query result to a client, and (3) *translation* of the retrieved query result into view-objects. We have seen other work for speeding up query processing, such as a high performance server utilizing parallelization, and we do not pursue the same work as in the PENGUIN project (Barsalou et al., 1990). Instead, we focus on the other two performance factors—transmission and translation. The key idea is to reduce the amount of redundant data that the system handles to instantiate view-objects.

Since the advent of the relational database, the universal method of query retrieval has been the *single flat relation* (SFR). The SFR method has the advantage of being able to apply the same relational query language uniformly on both base relations and query results, but is no longer useful in the view-object architecture because applications need a nested relation. A flat relation contains redundant duplicate subtuples inserted just to compose them into a "flat" relation. Their

numbers are in proportion to the cartesian products of join selectivities—rather than carrying any additional information, they just bring on the overhead of handling redundant data.

We present two alternatives to the SFR method. One retrieves a set of relation fragments (RFs) and the other retrieves a single-nested relation (SNR). RFs are materialized from base relations by reducing them with the selection, projection, and join operations as specified in the query. RFs contain all information required for restructuring them into an SNR. An SNR is a set of nested tuples in which an attribute can define another relation—called a *nested subrelation*. We develop the SFR, RF, and SNR methods and demonstrate that the RF and SNR methods are far more efficient than the SFR method in terms of both time and memory space.[1]

We assume that there are main memory databases[2] (Ammann et al., 1985; Bitton, 1986) on both the client and server sides. The case of main memory overflow is not considered. Note that the RF and SNR methods are less subject to memory overflow than the SFR method because they carry less redundant data. Here we emphasize that, while a main memory database is the environment that benefits most from the new methods, disk storage database systems benefit almost as well, according to sample case studies.

Following this introduction, we first provide a background framework that is useful for understanding the rest of the article. We describe the SFR, RF, and SNR methods in Section 3. In Section 4 we develop the cost models for the three methods and compare their costs in Section 5. The conclusion follows in Section 6.

2. Background Framework

We review relevant portions of the system model and introduce a *nesting format*. A full description of the model appears elsewhere (Lee, 1990; Lee and Wiederhold, 1994).

2.1 System Model

The system model has three elements: view-object types, views, and data. Figure 1 shows a schematic example of a view-object type and a view. A type defines the structure of view-objects. A view contains a relational query and defines a mapping between view-objects and relations. The data model uses the conventional relational model (Codd, 1970).

1. There must be some price for this. We may have to use two query processing frameworks (one on a client and one on a server) if we want to process view-objects further, because they are no longer flat relations.

2. A "main memory database" indicates that the entire database or an actively used subset of a database fits within main memory at the same time. As high density main memory chips become available at a lower cost, the number of applications running on main memory databases is increasing. According to Dill (1987), "approximately 50-75% of all disk accesses occur on data stored on 2-3% of the disk media"

Figure 1. Example of system model components

(a) A view-object type: (*'simple' denotes a simple attribute*).
Type O
[*Ao*: simple,
Bo: [*Do*: simple, *Eo*: simple,
. . . *Fo*: [*Ho*: simple, *Go*: simple]],
Co: [*Io*: simple, *Jo*: simple]]

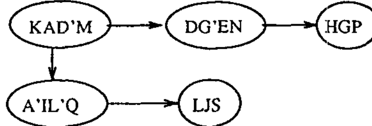
(b) Query part of a view:

- Query: (*Each relation name is made up of its attribute names*).

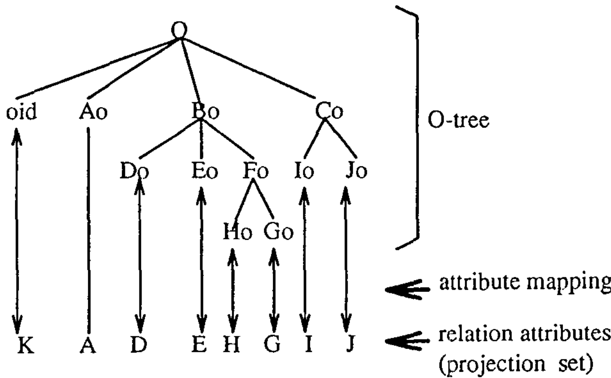
$$\pi_{KADEHGIJ}\left(\sigma_1 KAD'M \underset{D'\theta D}{\bowtie} \sigma_2 DG'EN \underset{G'\theta G}{\bowtie} \sigma_3 HGP \underset{A\theta A'}{\bowtie} \sigma_4 A'IL'Q \underset{L'\theta L}{\bowtie} \sigma_5 LJS\right)$$

where $\theta \in \{=, \neq, <, \leq, >, \geq\}$.

- Join tree (JT) of the query:



(c) Attribute mapping part of a view (OID denotes an object identifier).



2.2 View-object Type Model

A view-object type is defined as a tuple of attributes where each attribute is either a simple or complex attribute. A *simple* attribute has an atomic value or a set of atomic values. It is either internal or external to the object. An *internal* attribute has a primitive data type such as string or integer, while an *external* (or *reference*) attribute has another object type as its data type. The value of an external attribute is the identifier of a referenced object. A *complex* attribute defines an embedded object or a set of embedded objects.

We use value-oriented object identifiers (OIDs) (Khoshafian and Copeland, 1986; Abiteboul and Kanellakis, 1989) and retrieve them from the keys of relations.[3] Those relations providing OIDs are called *pivot relations* (Barsalou and Wiederhold, 1990; Barsalou et al., 1990; Law et al., 1991). An embedded object also has an associated OID which is mapped from the key of another relation. For instance, the embedded objects *Bo* and *Co* have hidden OIDs, which are not shown in Figure 1. As a result, there is more than one pivot relation, one for each OID. The OIDs of all *embedded* objects are needed only for mapping them to pivot relation keys and are not retrieved from the database. Not having an OID, an embedded object is not a "stand-alone" object.

We do not consider derived attributes for our view-object type. Derived attributes have no direct mapping to relation attributes and therefore are computed separately from relation attributes.

Given a view-object type *O*, we can build a tree (*O-tree*), defined as follows: (1) The root is labeled *O*; (2) A leaf is labeled as a simple attribute of the object *O* or its OID; (3) A non-leaf node is labeled by a complex attribute of the object *O*.

2.3 View Model

A view consists of two parts: a query part and a mapping part. For simplicity, we restrict queries to an acyclic select-project-conjunctive join query. Its join tree (JT) is rooted by the pivot relation. Two occurrences of the same relation are distinct. The mapping part in turn consists of an attribute mapping function (AMF) and a pivot description (PD).

AMF defines a mapping between view-object attributes and relation attributes. Because a view-object has no derived attribute, there is a *one-to-one* mapping between view-object attributes and relation attributes. Figure 1*c* shows an example between *O*'s attributes and relation attributes. There is a constraint on the definition of an AMF: two view-object attributes at the same level of an *O*-tree (e.g., *Do* and *Eo*) must be mapped to the relation attributes that belong to either the same relation or two different relations with one-to-one cardinality relationship.

PD consists of a set of pivot relations (PS) and a pivot mapping function (PMF). PMF defines a mapping between the keys of pivot relations and the OIDs of a view-object or its embedded objects. PS and PMF are irrelevant to the content of this article.

2.4 Nesting Format

A nesting format (Abiteboul and Bidoit, 1984) is the schema of a nested relation, and is generated from an O-tree and an AMF as follows: (1) Starting from the

3. Tuple identifiers are usable as well. Otherwise we assume that the system generates OIDs and maps them to the keys of corresponding relations.

root of the O-tree, recursively replace each node by the list of its children and (2) Replace each object attribute in the list with the relation attribute mapped by the AMF. For example, given the O-tree and AMF shown in Figure 1, we generate the nesting format *KA*(*DE*(*HG*))(*IJ*).

We can draw out the hierarchy of nested subrelations from a nesting format. The root of the tree represents a subrelation which is not nested within any other subrelation, and its descendents represent subrelations nested within their parents. We call such a tree a *nesting format tree* (NFT). In particular, the subrelation represented by the root is called a *pivot subrelation* because the root always contains an attribute which is mapped to an OID.

3. View-Object Instantiation Methods

We first give an overview of the SFR, RF, and SNR methods, and then give a detailed description of their steps. As will be explained, the SNR method is basically the same as the RF method except that the nesting step is carried out by a server. Therefore, we focus on the SFR and RF methods together and then discuss the SNR method separately.

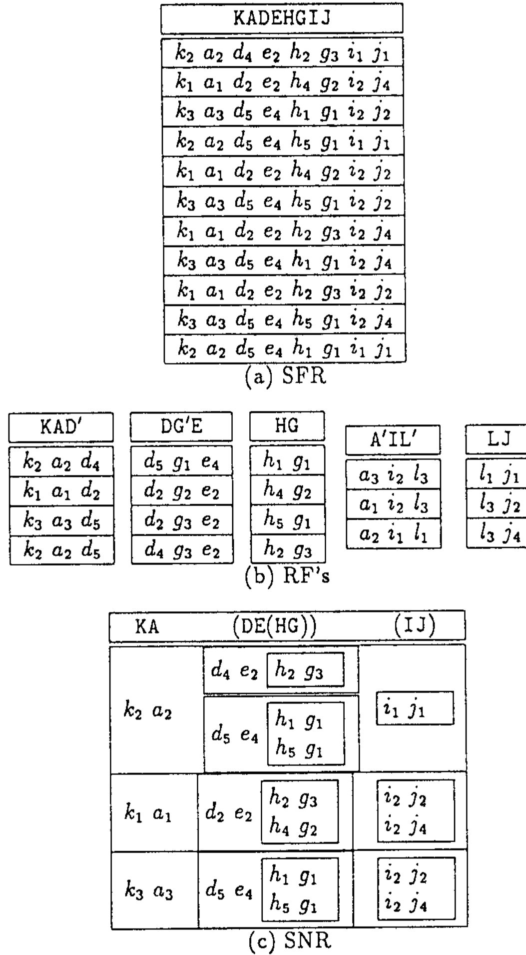
3.1 Overview of the Three Methods

The overall process is divided into three phases: *materialization*, *transmission*, and *translation*. In the SFR method, a query is materialized into an SFR by a server, transmitted as such, and translated into view-objects by a client. Translation is done in two steps: nesting, and reference resolution. The nesting step restructures a retrieved SFR into a nested relation. The reference resolution step resolves references among view-objects, thus configuring the retrieved view-objects into a network of references.

In the RF method, a query is materialized into a set of RFs by a server, transmitted as such, and translated into view-objects by a client. Translation is done in the same two steps as in the SFR method, but a different process is used for the nesting step due to the different structure of retrieved data. Because a client receives no separate information for linking tuples among the RFs, it creates the linkage information by building indexes on join attributes. Then, joins are performed starting from each tuple of the pivot RF and navigating along the joins to linked RFs. The result is an SNR, the same one that would be produced by the nesting step of the SFR method. The reference resolution step is the same as that of the SFR method.

In the SNR method, a query is materialized into an SNR, transmitted as such, and translated into view-objects by a client. A server first materializes a query using the RF method and then nests the query result into an SNR. We considered materializing a query directly into an SNR but did not take that approach because it impeded join reordering by a query optimizer (i.e., joins must be performed strictly

Figure 2. Example of SFR, RFs, and SNR



KADEHGIJ
$k_2\ a_2\ d_4\ e_2\ h_2\ g_3\ i_1\ j_1$
$k_1\ a_1\ d_2\ e_2\ h_4\ g_2\ i_2\ j_4$
$k_3\ a_3\ d_5\ e_4\ h_1\ g_1\ i_2\ j_2$
$k_2\ a_2\ d_5\ e_4\ h_5\ g_1\ i_1\ j_1$
$k_1\ a_1\ d_2\ e_2\ h_4\ g_2\ i_2\ j_2$
$k_3\ a_3\ d_5\ e_4\ h_5\ g_1\ i_2\ j_2$
$k_1\ a_1\ d_2\ e_2\ h_2\ g_3\ i_2\ j_4$
$k_3\ a_3\ d_5\ e_4\ h_1\ g_1\ i_2\ j_4$
$k_1\ a_1\ d_2\ e_2\ h_2\ g_3\ i_2\ j_2$
$k_3\ a_3\ d_5\ e_4\ h_5\ g_1\ i_2\ j_4$
$k_2\ a_2\ d_5\ e_4\ h_1\ g_1\ i_1\ j_1$

(a) SFR

KAD'
$k_2\ a_2\ d_4$
$k_1\ a_1\ d_2$
$k_3\ a_3\ d_5$
$k_2\ a_2\ d_5$

DG'E
$d_5\ g_1\ e_4$
$d_2\ g_2\ e_2$
$d_2\ g_3\ e_2$
$d_4\ g_3\ e_2$

HG
$h_1\ g_1$
$h_4\ g_2$
$h_5\ g_1$
$h_2\ g_3$

A'IL'
$a_3\ i_2\ l_3$
$a_1\ i_2\ l_3$
$a_2\ i_1\ l_1$

LJ
$l_1\ j_1$
$l_3\ j_2$
$l_3\ j_4$

(b) RF's

KA	(DE(HG))	(IJ)
$k_2\ a_2$	$d_4\ e_2$ [$h_2\ g_3$]; $d_5\ e_4$ [$h_1\ g_1$; $h_5\ g_1$]	$i_1\ j_1$
$k_1\ a_1$	$d_2\ e_2$ [$h_2\ g_3$; $h_4\ g_2$]	$i_2\ j_2$; $i_2\ j_4$
$k_3\ a_3$	$d_5\ e_4$ [$h_1\ g_1$; $h_5\ g_1$]	$i_2\ j_2$; $i_2\ j_4$

(c) SNR

in the nesting order). A client only has to do the reference resolution step, which is the same as in the other methods. Consequently, the SNR method is the same as the RF method except that the nesting step is done on a server. (The SNR method can be based on the SFR method, but it will be less efficient.)

Figure 2 shows an example of tuples obtained for each method by evaluating the query of Figure 1*b* with 'θ' $\equiv$ '='. These three methods have different sources of redundant data. An SFR contains *duplicate subtuples*, as discussed in the introduction. An RF contains no such duplicate subtuple. However, some RFs contain attributes that are not specified in the projection set of a query (e.g., *D', G', A', L'*). These attributes are materialized in, in addition to the projection set, and are needed to perform joins in the nesting step. We call them *extra join attributes* (EJAs). An SNR obviously contains fewer redundant subtuples than a corresponding SFR, but it still contains some subtuples duplicated in different nested subrelations. We call them

duplicate nested subtuples. We can make the following observations/hypotheses about their trade-offs: (1) The SNR method always carries fewer redundant data than the SFR method; (2) The RF method carries fewer redundant data than the SFR method, although there is a theoretical trade-off. (3) The amounts of redundant data in the RF and SNR methods are comparable.

Notation: Throughout this article, T denotes an SFR, F_i an RF, S_i a nested subrelation within an SNR, v_i a JT node, and u_i an NFT node. Note that there is a one-to-one mapping between $\{F_i\}$ and $\{v_i\}$, and between $\{S_i\}$ and $\{u_i\}$. We use two functions defining these one-to-one mappings—*RFJT* from $\{F_i\}$ to $\{v_i\}$ and *NSRNFT* from $\{S_i\}$ to $\{u_i\}$.

3.2 Materialization in the SFR and RF Methods

The materialization phase consists of two steps: *query processing* and *duplicate elimination*.

3.2.1 Query Processing. In main memory databases, the choice of query processing strategies (DeWitt et al., 1984, Bitton and Turbyfill, 1986; Shapiro, 1986; Lehman and Carey, 1986*b*; Bitton et al., 1987; Swami, 1989; Whang and Krishnamurthy, 1990) is based on the number of CPU cycles and memory space efficiency rather than the number of disk accesses and disk space efficiency. The results of comparing different query processing strategies obtained by some researchers (DeWitt et al., 1984; Lehman and Carey, 1986*b*; Shapiro, 1986) showed that hash-based query processing strategies are faster than others when large main memory is available. On the other hand, a main memory database system used in OBE (Bitton et al., 1987; Whang et al., 1987; Whang and Krishnamurthy, 1990) implemented a *pipelined nested loop join* with array indexes and obtained good performance in both time and memory space. One advantage of using this join algorithm is that it does not create intermediate relations during query processing.

Using the pipelined nested loop join strategy, the SFR query processing algorithm becomes as follows:

Algorithm 3.1 (SFR Query processing)
Input: base relations R_i, $i = 1,2,\cdots,n$; query
Output: SFR composite tuples.
For each $t_1 \in \sigma_1\ R_1$
    For each $t_2 \in \sigma_2\ R_2$ satisfying Φ_2
        $\ddots$
            For each $t_n \in \sigma_n\ R_n$ satisfying Φ_n
                Output $t_1.\pi_1 \parallel t_2.\pi_2 \parallel \cdots \parallel t_n.\pi_n$. /* $\parallel$ denotes a "concatenation." */

where σ_i denotes a selection condition on R_i, Φ_i denotes a conjunction of join predicates between R_i and each $R_1, R_2, \cdots, R_{i-1}$, and π_i denotes a subset of the projection set that comes from R_i.

For RF query processing, we modify Algorithm 3.1 to materialize a set of RFs instead of an SFR, rather than inventing a new algorithm. First, the single output statement must be decomposed into multiple statements (i.e., one output for each RF). Second, join attributes (η_i) should be materialized in addition to the projection set. Accordingly, the output statement is modified to "Output $t_1.(\pi_1 \cup \eta_1)$; $t_2.(\pi_2 \cup \eta_2)$; $\cdots$; $t_n.(\pi_n \cup \eta_n)$." Third, a tuple from an outer nested loop need not be emitted unless it is a new tuple. For example, $t_1 \in R_1$ in the outermost loop needs to be emitted only once for each completion of all the inner loops. We can use switches (sw_i's) for signaling whether a new tuple has been obtained from the outer loop in order to avoid these unnecessary emissions. These modifications result in Algorithm 3.2.

Algorithm 3.2 (RF Query processing)
Input: base relations $R_i, i = 1, 2, \cdots, n$; query
Output: RFs $F_i, i = 1, 2, \cdots, n$.

```
For each t1 ∈ σ1 R1,
    Set sw1.
    For each t2 ∈ σ2 R2 satisfying Φ2,
        Set sw2.
        ⋱
            For each tn ∈ σn Rn satisfying Φn,
                Set swn.
                For each swi, i = 1, 2, ··· ,n,
                    If swi is set then begin
                        Output ti.(πi ∪ ηi).
                        Reset swi.
                    end
```

By comparing Algorithms 3.1 and 3.2, we see that both execute the same nested loops and take approximately the same time. However, they differ in the amount of data emitted by the output statements. In Algorithm 3.1, an SFR composite tuple is emitted for every execution of the innermost loop, whereas in Algorithm 3.2, an RF tuple is emitted only if all inner loops are completed. Therefore, the RF method emits fewer data than the SFR method.

3.2.2 Duplicate Elimination. An SFR or RF produced by the query processing step may have duplicate tuples. These eventually result in duplicate view-objects, which are difficult to manage by applications. Therefore, duplicate tuples are removed beforehand by either sorting or hashing. We use hashing because it is usually faster and its result can be pipelined to the transmission step (not for sorting).

We use a simple chained bucket hashing (Knuth, 1973) for which the bucket header is an array of pointers to buckets and each chained bucket is a record of a hashed tuple and a pointer to the next bucket. Given this structure, the algorithm for eliminating duplicates in pipelining with transmission becomes as follows:

Algorithm 3.3 (Duplicate elimination)

1. Allocate a hashing bucket header.
2. For each tuple t_o emitted from the query processing,
   (a) Compute a hashed address $h(t_o)$. (h: a hashing function)
   (b) For each bucket in the chain starting at $h(t_o)$,
   If $t_o = t_b$ then continue step 2. (t_b: the tuple in the bucket)
   (c) Insert a new bucket containing t_o into the chain and transmit t_o. /* t_o is new. */

3.3 Translation in the SFR and RF Methods

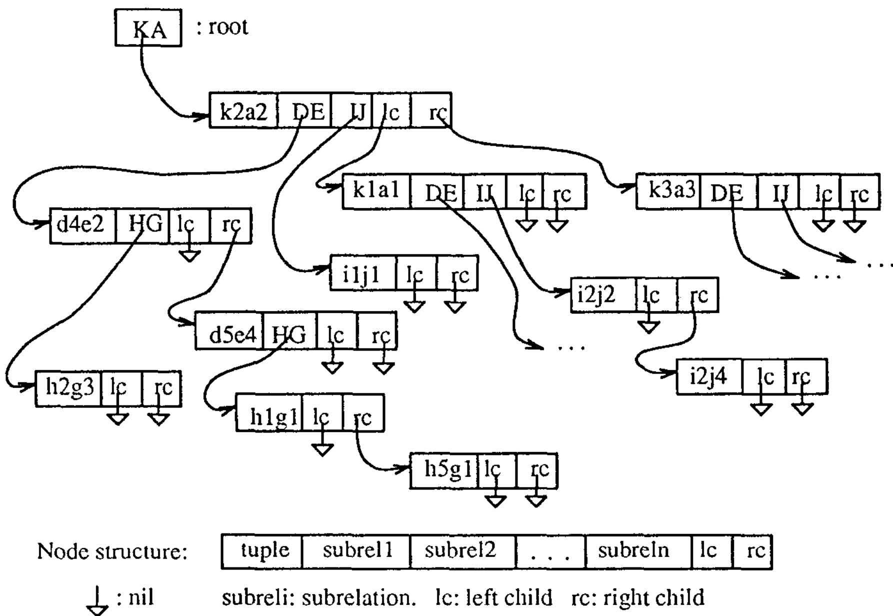
As mentioned in Section 3.1, the translation phase has two steps: *nesting* and *reference resolution.* The nesting step is carried out differently in the SFR and RF methods. In the SFR method, it is done by decomposing received composite tuples into subtuples corresponding to different nested subrelations and assembling the decomposed subtuples into nested tuples. In the RF method, it is done by *creating indexes* on the join attributes of the RFs and performing *navigational joins.* Navigation starts from the pivot RF and follows the join links to find matching tuples in the RFs. Found matching tuples are assembled into nested tuples according to an *assembly plan*, which is generated by comparing a JT and an NFT. Before the index creation, one arbitrary join predicate is selected from each conjunction of join predicates in the *join purge* step. The reference resolution step is out of our scope because its process is specific to the view-object schema defined by the application. Besides, omitting this step does not affect the cost comparison result because its process is identical in all three methods.

3.3.1 Structure of an SNR. Subtuples decomposed by an SFR may have duplicates, even though the composite tuples do not. RFs may have duplicate tuples as well after being stripped of EJAs (with projections). Therefore, every insertion into an output SNR must be preceded by a searching for duplicates, and consequently searchings are performed more frequently than insertions in the nesting step. (It is more manifest for an SFR.) This leads to the fact that the structure of an SNR should show good *searching* performance.

Figure 3 shows the structure of an SNR we used. Each nested subrelation is implemented as a binary search tree (BST). The top-most root (*KA*) contains a pointer to the BST of the pivot subrelation, and each node of a BST contains a tuple, pointers to the nested BSTs, and pointers to its left child and right child. Searching or insertion of a tuple takes $O(\log_2 N)$ time for each BST, where N is the number of tuples in a BST.

3.3.2 Nesting of an SFR. Fischer and Thomas (1983) introduced NEST as an operator for restructuring a flat relation into a nested relation. Similar concepts were also discussed by Abiteboul and Bidoit (1984) and Roth et al. (1988). Our nesting

Figure 3. Structure of an SNR



process is an instance of implementing the NEST operator. Figure 3 shows the SNR after inserting the first three SFR tuples of Figure 2*a*. SFR nesting can be performed pipelined with the reception of data from a server.

Algorithm 3.4 (SFR Nesting)
Input: received SFR tuples $\{t_r\}$; NFT.
Output: SNR.

1. Allocate an empty (root only) SNR.
2. w_p := the root of the empty SNR.
3. u_p := the root of NFT.
4. For each t_r, Assemble(w_p,u_p,t_r).

where Assemble(w_p,u_p,t_r) inserts a decomposed subtuple $\pi_{u_p}t_r$ into a BST pointed by $w_p.u_p$.

Algorithm 3.5 (SFR Assemble)
Input: SNR node w_i; NFT node u_i; composite tuple t_r.
Output: SNR with t_i inserted if t_i is new.

1. $t_i := \pi_{u_i}t_r$ /* Project t_r on u_p. */
2. w_r := the BST node pointed by $w_i.u_i$. /* w_r is the root of a BST to be searched. */

3. If (w_c := Search(w_r,t_i)) = NOT_FOUND then Insert-tuples(w_i,u_i,t_i)
   else /* t_i already exists. */
   (a) Ψ := the set of u_i's children (u_c) in NFT.
   (b) If Ψ = { } then return
       else for each $u_c \in \Psi$, Assemble(w_c,u_c,t_r).

where Search(w_r,t_i) finds a node containing t_i from the BST rooted by w_r; and Insert-tuples(w_i,u_i,t_i) inserts a tuple t_i into the BST pointed by $w_i.u_i$, and recursively inserts all nested subtuples of t_i (corresponding to u_i's descendents in the NFT).

Algorithm 3.6 (Search)
Input: SNR node (w_i); tuple t_i to be searched for.
Output: return NOT_FOUND or the found node.
    If w_i = nil then return NOT_FOUND
    else if w_i.tuple = t_i then return w_i
    else if (w_i.tuple < t_i) then return Search(w_i.lc, t_i)
    else return Search(w_i.rc, t_i).

Algorithm 3.7 ([Insert-tuples)
Input: SNR node w_i; NFT node u_i; tuple t_i to be inserted.
Output: SNR with t_i inserted.

1. Allocate an empty node w_m and copy t_i to w_m.tuple.
2. w_e := Insert(w_i,u_i,w_m). /* Insert t_i. */
3. /* Insert t_i's nested subtuples. */
   Ψ := the set of u_i's children in the NFT.
   If Ψ = { } then return
   else for each $u_c \in \Psi$, Insert-tuples(w_e,u_c,$t_r.u_c$).

where Insert(w_i,u_i,w_m) inserts a new node w_m into the BST pointed by $w_i.u_i$ and returns the inserted node.

Algorithm 3.8 (Insert)
Input: SNR node w_i; NFT node u_i; new node w_m.
Output: return the inserted node.
    If $w_i.u_i$ = nil then return $w_i.u_i$:= w_m /* Insert w_m. */
    else if t_i < $w_i.u_i$.tuple then Insert($w_i.u_i.lc$, u_i, w_m)
    else Insert($w_i.u_i.rc$, u_i, w_m).

3.3.3 Nesting of an RF. Nesting of retrieved RFs is performed in four steps: join purge, assembly planning, index creation, and navigational join.

Join Purge. In the join purge step, a conjunction of join predicates in a query is reduced to a single join predicate by choosing one of them arbitrarily.[4] This join reduction does not affect the result of the nesting step, as verified by the following theorem.

Theorem 3.1 Let us consider a conjunctive join predicate $A_1\theta_1\, B_1 \wedge A_2\theta_2\, B_2 \wedge \cdots \wedge A_n\theta_n\, B_n$ between two RFs F_1 and F_2 retrieved from a server. Then, for an arbitrary pair of tuples $\langle\, t_1 \in F_1,\, t_2 \in F_2 \rangle$,

$$(t_1.A_1\theta_1 t_2.B_1) \wedge (t_1.A_2\theta_2 t_2.B_2) \wedge \cdots \wedge (t_1.A_n\theta_n t_2.B_n) \qquad (1)$$

if and only if

$$t_1.A_i\theta_i t_2.B_i \text{ for some } i \in [1, n] \qquad (2)$$

Proof: Since the "only if" part is obvious, we prove only the "if" part: Let us assume Equation 1 is not satisfied and Equation 2 is satisfied. Then, there is at least one $j \in [1,n]$ such that $j \neq i$ and $\neg(t_1.A_j\theta_j\; t_2.B_j)$. However, if $t_1.A_j\theta_j\; t_2.B_j$ is false, $t_1 \notin F_1$ if $t_2 \in F_2$ and $t_2 \notin F_2$ if $t_1 \in F_1$ by the definition of join. It contradicts with the given assumption that $t_1 \in F_1$ and $t_2 \in F_2$. □

Assembly Planning. In this step, we prepare a plan to assemble the tuples that will be collected by navigational joins. An assembly plan (AP) is a transformation from JT nodes ($\{v_i\}$) to NFT nodes ($\{u_i\}$). Figure 4 illustrates it for the view-object shown in Figure 1. An NFT node is obtained from one or more JT nodes via relational projections and joins. A JT node represents an RF, while an NFT node represents a nested subrelation of an SNR. Joins are needed only if the schema of an NFT node is not a subset of the schema of any RF but spans the schemas of two or more RFs. The *IJ* node of the NFT in Figure 4 is such a case. It is merged from the JT nodes $A'IL'$ and LJ via a join and projection. Any merged JT nodes (i.e., RFs) always have a one-to-one cardinality relationship.

An AP is represented by a set of expressions of the following form:

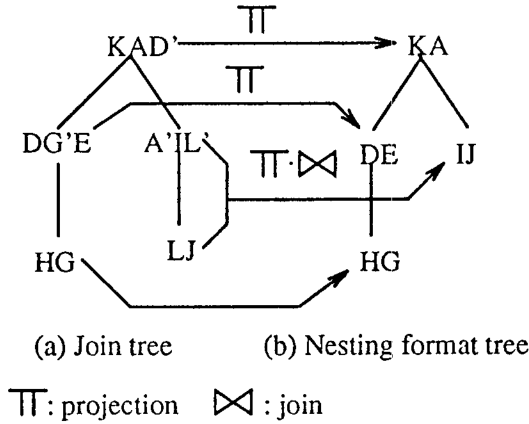
$$u := \pi_u(v_1 \bowtie v_2 \cdots \bowtie v_k)$$

The following example shows the assembly plan (name this *AP-1*) for the JT and NFT of Figure 4: $\{KA := \pi_{KA}KAD', DE := \pi_{DE}DG'E, HG := HG, IJ := \pi_{IJ}(A'IL' \underset{L'\theta L}{\bowtie} LJ)\}$. We use the same denotations (v_i, u_i, and AP) for the schema and tuples of JT and NFT nodes. For example, AP($A'IL'$, LJ) returns IJ and AP($a_3i_2l_3$, l_3j_2) returns i_2j_2.

The algorithm for generating an AP is as follows:

4. It will be more practical to select one that is easy to compute, such as an equijoin between integer attributes.

Figure 4. Example of an assembly plan



Algorithm 3.9 (Assembly planning)
Input: JT; NFT.
Output: AP.

1. For each node v newly visited while traversing JT, starting from the root,
   (a) Find an NFT node u such that $u \subseteq v$.
   (b) If found then
      i. If $u = v$ then add "$u := v$" to AP
         else add "$u := \pi_u\, v$" to AP
      ii. Mark v as "visited."

   else
      i. Find the nodes $\{v_1, v_2, \cdots, v_k\}$ of a *minimal* subtree of JT rooted by v such that for some NFT node u, $u \subseteq v_1 \cup v_2 \cup \cdots \cup v_k$.
      ii. Add "$u := \pi_u(v_1 \bowtie v_2 \cdots \bowtie v_k)$" to the AP.
      iii. Mark $v_1, v_2, \cdots, v_k$ as "visited."

Index Creation. Once redundant joins are removed, indexes are created on the join attribute of each RF except the pivot RF. According to the performance study by Lehman and Carey (1986*a*) a chained bucket hashing gives the best performance among all main memory index operations except for a range query. Because we do not need a range query, bucket hashing is appropriate for our use. The index is composed of a bucket header table and chained buckets linked to each header. Unlike the example in Section 3.2.2, each bucket header and chained bucket contains a *pointer* to a tuple instead of an actual tuple. An index organized this way shows the best storage cost/performance ratio when its bucket header table contains approximately *half* the number of indexed tuples (Lehman and Carey, 1986*a*). The algorithm for creating an index is as follows:

Algorithm 3.10 (Index creation)
Input: RF F_i; join attribute A_i of F_i.
Output: a chained bucket hashing index on the attribute A_i of F_i.

1. Allocate a bucket header table.
2. For each value a of $F_i.A_i$,
   (a) Compute the hashed address $h(a)$. (h: hashing function)
   (b) Insert a new bucket containing a at the hashed address $h(a)$ (without duplicate checking).

Index creation cannot start until the entire tuples of all RFs are received because (1) a hashing index requires the number of indexed tuples to be known before an index is created and (2) the tuples of RFs are received in row-wise order (i.e., different tuples from different RFs are intermixed).

Navigational Join. Once indexes are created and an assembly plan is prepared, we perform navigational joins on the RFs staring from the pivot RF and following the index paths. There are always one or more matching tuples because non-matching tuples have already been discarded in the materialization step. The set of matching tuples thus found are assembled into nested tuples according to the assembly plan. For example, starting from the third tuple [k_3 a_3 d_5] of KAD' in Figure 2*b*, we find the following set of matching tuples from the other RFs: [d_5 g_1 e_4] from $DG'E$, [h_1 g_1], [h_5 g_1] from HG, [a_3 i_2 l_3] from $A'IL'$, and [l_3 j_2], [l_3 j_4] from LJ. These tuples are assembled into the last nested tuple of Figure 2*c* with the assembly plan *AP-1* (Section 3.3.3). The following algorithms describe this procedure more rigorously.

Algorithm 3.11 (Navigational join)
Input: F_i's (F_1 is the pivot RF); JT; NFT; AP.
Output: SNR.

1. Allocate an empty SNR.
2. w_p := the root of the empty SNR.
3. u_p := the root of NFT.
4. For each tuple $t_p \in F_1$, Assemble(w_p,u_p,t_p).

Assemble(w_p,u_p,t_p) starts navigation from t_p and collects a set of matching tuples from F_i, $i = 2,3,\cdots,n\}$. Then, for each set of matching tuples, it finds an associated expression from the AP and executes the expression on the tuples. The resulting tuples are inserted into the SNR.

Algorithm 3.12 (RF Assemble)
Input: SNR node w_i; NFT node u_i; tuple t_0 from which to start navigation.
Output: SNR with newly inserted tuples.

1. w_r:=the node pointed by $w_i.u_i$./*w_r is the root of a BST to be searched.*/

2. Find $\{v_1,v2,\cdots,v_k\}$ from AP such that $u_i = \text{AP}(v_1,v_2,\cdots,v_k)$.
   /* $k > 1$ if and only if a merging is required. */

3. /* For $i = 1,2,\cdots,k$, let F_i be $\text{RFJT}^{-1}(v_i)$, and Φ_i be the join predicate between F_i and F_j where RFJT(F_j) is the parent of RFJT(F_i) in the JT. */
   For each $t_1 \in \text{Match}(t_0,F_1,\Phi_1)$,
   For each $t_2 \in \text{Match}(t_1,F_2,\Phi_2)$,
   $\ddots$
   For each $t_k \in \text{Match}(t_{k-1},F_k,\Phi_k)$,
   (a) $t_c := \text{AP}(t_1,t_2,\cdots,t_k)$. /* Execute the assembly plan. */
   (b) If ($w_c := \text{Search}(w_r,t_c)$) = NOT_FOUND then $w_c := \text{Insert}(w_i,u_i,t_c)$.
   (c) $\Psi :=$ the set of u_i's children in NFT.
   (d) If $\Psi = \{\ \}$ then return
       else for each $u_c \in \Psi$, Assemble(w_c,u_c,t_c).

where Search and Insert are the same as Algorithms 3.6 and 3.8. No duplicate checking is necessary for an insertion unless a projection is prescribed in the AP. Given a tuple $t_i \in F_i$, Match(t_i,F_j,Φ_j) finds matching tuples from F_j through an index built for the join predicate Φ_j.

Algorithm 3.13 (Match)
Input: $t_i \in F_i$; F_j; join predicate "$F_i.A\theta\ F_j.B$."
Output: $\{t_j |\ t_j \in F_j,\ t_i.A\theta\ t_j.B\}$.

1. Compute the hashed address $h(t_i.A)$. (h: hashing function)

2. For each bucket from the bucket header through the end of the chain,
   If $t_i.A\theta\ t_j.B$ then collect t_j. (t_j: a tuple pointed by the bucket entry.)

3.4 The SNR Method

Because the SNR method is based on the RF method, we focus only on the modifications needed to adapt the RF method to the SNR method. Query processing and duplicate elimination are exactly the same as in the RF method, except that emitted tuples are written to an output buffer instead of being transmitted to a client. Once the tuples of all RFs are collected in the output buffer, they are converted into an SNR on a server using the same steps as in the RF method. The navigational join step needs to be modified so that matching tuples are not only assembled into nested tuples, but also transmitted to a client. According to Algorithm 3.12, the tuples of nested subrelations are transmitted in a depth-first search order of an NFT. Delimiters are needed to distinguish between the tuples of different nested subrelations. For example, the data stream transmitted for the SNR of Figure 2c looks as follows. ("⟨" and "⟩" are delimiters.)

$$\langle \text{ KA } \langle \text{ DE } \langle \text{ HG } \rangle\rangle\langle \text{ IJ } \rangle\rangle\langle\, k_2a_2\langle\, d_4e_2\langle\, h_2g_3\rangle\rangle\langle\, d_5e_4\langle\, h_1g_1\ h_5g_1\rangle\rangle\langle\, i_1j_1\rangle\rangle$$
$$\langle\, k_1a_1\langle\, d_2e_2\langle\, h_2g_3\ h_4g_2\rangle\rangle\langle\, i_2j_2\ i_2j_4\rangle\rangle\ \langle\, k_3a_3\langle\, d_5e_4\langle\, h_1g_1\ h_5g_1\rangle\rangle\langle\, i_2j_2\ i_2j_4\rangle\rangle$$

where $\langle KA\langle DE\langle HG\rangle\rangle\langle IJ\rangle\rangle$ is a header describing the format of the following data stream. A data stream is composed of segments. A segment contains the tuples that will belong to the same nested subrelation when assembled into an SNR. The above example shows three segments starting with k_1a_1, k_2a_2, and k_3a_3, respectively.

A client has only to parse the received data stream and assemble the extracted tuples into an SNR. Algorithm 3.14 describes the assembly process. For each tuple t_i read from the data stream, t_i is inserted as a nested subtuple of the previous tuple if t_i is preceded by "$\langle$." Otherwise, t_i is inserted in the same nested subrelation as in the previous tuple. w_c denotes the currently inserted node and w_p denotes the previously inserted node. They are moved one level up for each "$\rangle$." Super(w_p) returns the node in which w_p is nested.[5]

Algorithm 3.14 (SNR Assemble)
Input: formated stream of SNR tuples; NFT.
Output: assembled SNR.

1. Allocate an empty SNR.
2. w_c := the root of the empty SNR.
3. For each item d read from the data stream,
   - If d = "$\langle$" then $w_p := w_c$.
   - If $d = t_i$ (a tuple) then
     Find the schema S_i of t_i from the header.
     u_p := NSRNFT(S_i).
     w_c := Insert(w_p,u_p,t_i).
   - If d = "$\rangle$" then $w_c := w_p$; w_p := Super(w_p).

where Insert is the same as Algorithm 3.8. Note that we need no searching before an insertion because duplicates have already been eliminated on a server.

4. Cost Model

4.1 A Platform for Cost Modeling

It is too complicated a task to obtain a cost model of main memory-resident operations because the cost depends on so many factors (e.g., hardware, programming language, programming style, and system load). Since our purpose is *comparing* costs as opposed to estimating them, we make some simplifications in the cost

5. To implement this function, we need to keep both back-pointers to previous nodes or a chain of inserted nodes.

models without affecting the comparison results. First, the cost items that are common to all three methods are excluded. These are the costs of the query processing and reference resolution steps. Second, we exclude the cost of accessing schema information, which is negligible compared with the cost of operations on data tuples. Third, we ignore the difference between server speed and client speed. Their effect on the cost comparison result is marginal, particularly in an environment with significant network communication overhead.

We use only the execution time as the measure of cost—although required main memory space is another important measure—because there is no trade-off between time and space in our case. The total cost is the sum of local processing cost and transmission cost. Local processing cost is the total execution time spent on a server and a client. Transmission cost is the time for sending a query result to a client.

We consider only complex queries (i.e., queries with one or more joins). SFR, RF, and SNR methods become identical if a query is a simple query (i.e., it has no join): The base relation specified in a simple query is reduced to a single fragment, transmitted to a client, and linked to other view-objects through the reference resolution step. The nesting step is not needed for the single fragment.

4.1.1 Cost and Data Parameters. Table 1 shows the cost parameters for elementary main memory and network communication operations. They were measured on a SUN-3 workstation, between two SUN-3s on the same Ethernet LAN, and between a SUN-3 on the Stanford campus and another SUN-3 on the University of Illinois campus. We used *CPU time* for main memory operations because it is quite insensitive to the system load, whereas we used *elapsed time* for network communication operations because most communication time is spent on the network. Table 2 shows the data parameters of an SFR, RFs, and an SNR.

4.1.2 Alternative Data Parameters: α_{ij} and β_{ij}. We define α_{ij} as the domain selectivity (i.e., the average number of tuples with the same value) of F_j's join attributes. Then, α_{ij} is related to N_{f_j} and $D_{f_{ij}}$ as follows:

$$\alpha_{ij} = \frac{N_{f_j}}{D_{f_{ij}}} \tag{3}$$

Because all non-matching tuples of RFs have already been discarded in the query materialization step, $D_{f_{ji}} = D_{f_{ij}}$. Hence, α_{ij} can be interpreted as the average number of matching tuples in F_j for each tuple of F_i. We call α_{ij} a *selectivity from* F_i *to* F_j. Since $D_{f_{ji}} = D_{f_{ij}}$, the following is always true:

$$N_{f_j} \leq N_{f_i}\alpha_{ij} \tag{4}$$

where the equality holds if and only if $D_{f_{ji}} = N_{f_i}$ (i.e., F_i's join attributes have unique values).

Table 1. Cost parameters

(a) Main memory cost parameters (CPU time)

Parameter	Description	Value
C_{bs}	Cost of elementary binary search operation (compare and move left or right)	19 μsec
C_{cm}	Cost of comparing two tuples	9.2 μsec
C_{ci}	Initial cost of copying a tuple	11 μsec
C_{cb}	Per-byte cost of copying a tuple	0.17 μsec/byte
C_e	Cost of evaluating a join predicate (equijoin on integer attributes)	16 μsec
C_{fl}	Per-byte cost of folding tuple into integer	0.92 μsec/byte
C_{hc}	Cost of computing hashed address for integer hashing key	9.5 μsec
C_{ma}	Cost of allocating memory within workspace	1.2 μsec
C_{mp}	Cost of moving (reading or writing) pointer	0.88 μsec
C_{pi}	Initial cost of performing projection on tuple	4.3 μsec
C_{pb}	Per-byte cost of performing projection on tuple	1.1 μsec/byte
C_{si}	Initial cost of computing integer hashing key from a scanned relation column	17 μsec
C_{sn}	Per-tuple cost of computing integer hashing key from a scanned relation column	14 μsec/tuple

(b) Communication cost parameters (elapsed time)

Parameter	Description	Value	
		LAN	WAN
C_l	Latency of sending a message	2.5 msec	53 msec
C_b	Per-byte data transmission cost	3.4 μsec/byte	60 μsec/byte

β_{ij} is defined as the *average degree of nesting*, which is the average number of tuples in S_j for each tuple of S_i where S_j is an immediate nested subrelation of S_i. Put in another way,

$$\beta_{ij} = \frac{N_{s_j}}{N_{s_i}} \tag{5}$$

Note that $\beta_{ij} \geq 1$.

Table 2. Data parameters

Parameter	Description
SFR (T)	
N_t	Cardinality after duplicate elimination
d_t	Ratio between cardinalities before and after duplicate elimination ($0 < d_t \leq 1$)
T_t	Tuple size
RF (F_i,i = 1,2,$\cdots$,n_f where F_1 is the pivot RF)	
n_f	Number of RFs ($n_f > 1$ for complex queries)
N_{f_i}	Cardinality of F_i after duplicate elimination
d_{f_i}	Ratio between cardinalities of F_i before and after duplicate elimination. ($0 < d_{f_i} \leq 1$)
$D_{f_{ij}}$	Domain cardinality (i.e., number of distinct values) of F_is join attribute for join between F_i and F_j
T_{f_i}	Tuple size of F_i
ρ_{f_i}	Extra join attribute (EJA) ratio (i.e., ratio between size of EJAs in F_i and T_{f_i}. ($0 \leq \rho_{f_i} \leq 1$)
SNR (S_i,i=1,2,$\cdots$,n_s where S_1 is pivot nested subrelation)	
n_s	Number of nested subrelations in an SNR
N_{s_i}	Cardinality of S_i
T_{s_i}	Tuple size of S_i

4.2 Derivation of Cost Formulas

In this section we develop the cost formulas of all but the query processing and reference resolution steps. The following short-hand notations are used in the cost formulas.

$$C_{colscan}(N) = C_{si} + C_{sn}N \text{ for scanning } N \text{ tuples} \quad (6)$$

$$C_{copy}(T) = C_{ci} + C_{cb}T \text{ for copying tuple of size } T \text{ bytes} \quad (7)$$

$$C_{project}(T) = C_{pi} + C_{pb}T \text{ for projecting subtuple of size } T \text{ bytes out of tuple} \quad (8)$$

4.2.1 Duplicate Elimination Cost. The duplicate elimination process is the same for all three methods except that it is applied to different structures. We make the following two assumptions for hashing tuples (Algorithm 3.3): (1) We allocate as many bucket headers as half the cardinality of a hashed relation, which can be estimated by a query optimizer. (Otherwise, we could use a linear hashing); (2)

The shift folding technique (Mauer and Lewis, 1975; Horowitz and Sahni, 1976) is used for the hashing of tuples (a tuple is divided into integer parts, which are then added to obtain an integer hashing key).

Let N be the relation cardinality after duplicate elimination, T be the tuple size, and d be the ratio of the cardinalities before and after duplicate elimination ($0 < d \leq 1$). The allocation of a bucket header costs C_{ma}. Step 2 of Algorithm 3.3 is repeated N/d times. The cost of computing a hashed address is computed as a function of T as follows.

$$C_{tuphash}(T) = C_{fl}T + C_{hc} \tag{9}$$

Among the N/d hashed tuples, N tuples are actually inserted and the other N/d - N tuples are discarded. If the same tuple already exists, it takes the cost of traversing an average half of a bucket chain, C_{mp} + $(N_b/2)(C_{cm}+C_{mp})$ where N_b is the number of buckets that has been inserted in the chain so far. Otherwise, it costs traversing the entire bucket chain (C_{mp} + $N_b(C_{cm}+C_{mp})$) and inserting a new bucket in the chain ($C_{ma}+C_{copy}(T)$ + 2 C_{mp}).

N_b is obtained as follows. N hashed entries are inserted in $N/2d$ bucket headers. If $N > N/2d$, all bucket headers are filled, assuming that the hash function distributes a hashing key uniformly over the bucket header table. In this case, the ultimate value of N_b becomes $N/(N/2d)$ = 2d. Otherwise, only N bucket headers out of $N/2d$ headers are filled and the ultimate value of N_b becomes 1. Using half the ultimate values as expected values,

$$N_b = \text{MAX}\,(d, \tfrac{1}{2}) \tag{10}$$

The cost of inserting a hashed tuple into the hash is computed as a function of T and d as follows. (The cost of transmitting the inserted tuple is part of the transmission cost and is not included here.)

$$\begin{aligned} C_{tupinsert}(d,T) &= d(C_{mp} + N_b(C_{cm} + C_{mp}) + C_{ma} + C_{copy}(T) + \\ &\quad 2C_{mp}) + (1-d)(C_{mp} + \frac{N_b}{2}(C_{cm} + C_{mp})) \end{aligned} \tag{11}$$

Using Equations 9 and 11, the SFR duplicate elimination cost is

$$C_{sfrde} = C_{ma} + \frac{N_t}{d_t}(C_{tuphash}(T_t) + C_{tupinsert}(d_t, T_t)) \tag{12}$$

and for all RFs it is computed as follows.

$$C_{rfde} = \sum_{i=1}^{n_f}(C_{ma} + \frac{N_{f_i}}{d_{f_i}}(C_{tuphash}(T_{f_i}) + C_{tupinsert}d_{f_i}, T_{f_i}))) \tag{13}$$

Because SNR query processing also produces RFs, SNR duplicate elimination incurs the same cost as the RF method except for the cost of writing non-duplicate

tuples to an output buffer. This cost for each RF is $C_{copy}(T_{f_i})N_{f_i}$. Thus, the total cost is:

$$C_{snrde} = C_{rfde} + \sum_{i=1}^{n_f} C_{copy}(T_{f_i})N_{f_i} \tag{14}$$

4.2.2 Nesting Cost.

Binary Search Tree Searching and Insertion Costs. The searching (Algorithm 3.6) and insertion (Algorithm 3.8) of one tuple are used commonly for all three methods and therefore we derive their cost formulas separately here. We assume that the binary search trees (BSTs) implementing nested subrelations are well-balanced.[6] Let M be the number of tuples that are to be inserted into a BST. Every insertion attempt requires one searching to check out duplicates. Let N denote the number of tuples that are actually inserted into a BST. According to Knuth (1973), a single searching requires about 1.386 $\log_2 k$ comparisons (k is the number of nodes currently in the BST) for a well-balanced BST, considering both a successful and an unsuccessful search. If we assume that the insertions of the N tuples out of M tuples occur at regular intervals, the value of k is incremented at every M/N insertion attempt. The total searching cost for inserting N tuples out of the attempted M tuples then is computed as follows:

$$C_{binsearch}(M,N) = \sum_{k=1}^{N}(\frac{M}{N}1.386C_{bs}\log_2 k) \tag{15}$$

Insertion cost is the sum of the costs of searching for a node unsuccessfully and inserting it as a leaf of the BST. An unsuccessful search of a BST requires $\log_2(k+1)$ comparisons. Insertion at a leaf requires allocating an empty node (C_{ma}), copying a tuple into it ($C_{copy}(T)$), and writing a pointer to it (C_{mp} in its parent node). Thus, the total cost of inserting N tuples into a BST is computed as follows:

$$C_{bininsert}(N,T) = \sum_{k=1}^{N}(C_{bs}\log_2(k+1) + C_{ma} + C_{copy}(T) + C_{mp}) \tag{16}$$

There will be N_{s_i} tuples inserted into a nested subrelation S_i of the final output SNR. Let $S_{par(i)}$ denote the nested subrelation such that NSRNFT($S_{par(i)}$) is the parent of NSRNFT(S_i). Then, there are $N_{s_{par(i)}}$ BSTs implementing S_i (i.e., one BST for each tuple of $S_{par(i)}$). Let M_{s_i} denote the number of tuples that are attempted for an insertion into S_i. If we assume that tuples are uniformly distributed into every BST of S_i, $M_{s_i}/N_{s_{par(i)}}$ tuples are attempted for an insertion

6. In fact, well-balanced trees are common, and degenerate trees are very rare (Knuth, 1973). Even if a BST must be balanced sometimes, a tree balancing involves only pointer movements and incurs negligible cost.

and $N_{s_i}/N_{s_{par(i)}}$ tuples are actually inserted into each BST of S_i. Thus, the total cost of inserting N_{s_i} tuples into S_i out of the attempted M_{s_i} tuples is computed as follows:

$$C_{sisearch}(M_{s_i}, N_{s_i}, N_{s_{par(i)}}) = N_{s_{par(i)}} C_{binsearch}(\frac{M_{s_i}}{N_{s_{par(i)}}}, \frac{N_{s_i}}{N_{s_{par(i)}}}) \quad (17)$$

$$C_{siinsert}(N_{s_i}, T_{s_i}, N_{s_{par(i)}}) = N_{s_{par(i)}} C_{bininsert}(\frac{N_{s_i}}{N_{s_{par(i)}}}, T_{s_i}) \quad (18)$$

SFR Nesting Cost. We consider only the costs of projecting, searching (Algorithm 3.6), and inserting tuples (Algorithm 3.7), which are operations on data tuples and whose costs are dominant.

According to Algorithm 3.4, N_t composite tuples are decomposed into subtuples of $S_1, S_2, \cdots, S_{n_s}$ by projections and assembled into an SNR. For each subtuple of S_i, projecting it from a composite tuple costs $C_{project}(T_{s_i})$, searching for it from S_i costs $C_{sisearch}(N_t, N_{s_i}, N_{s_{par(i)}})$, and inserting it into S_i costs $C_{siinsert}(N_{s_i}, T_{s_i}, N_{s_{par(i)}})$. Hence, the total cost is computed as follows:

$$\begin{aligned} C_{sfrnest} &= \sum_{i=1}^{n_s}(C_{project}(T_{s_i})N_t + C_{sisearch}(N_t, N_{s_i}, N_{s_{par(i)}}) + \\ &\quad C_{siinsert}(N_{s_i}, T_{s_i}, N_{s_{par(i)}})) \end{aligned} \quad (19)$$

RF Nesting Cost. We ignore the costs of the join purge step and the assembly planning step because they are not operations on data tuples. Accordingly, we approximate the RF nesting cost as the sum of the index creation cost and the navigational join cost.

$$C_{rfnest} \approx C_{ixcrt} + C_{navjn} \quad (20)$$

The number of RF joins is always one less than the number of the RFs (i.e., $n_f - 1$) after the join purge step.

Index creation (Algorithm 3.10): A bucket header allocation costs C_{ma}. The linear scan of F_i costs $C_{colscan}(N_{f_i})$. We assume all join attributes are integers so that no folding is required. A hashing computation costs C_{hc}. An insertion to a hashing bucket chain takes the cost of allocating a bucket (C_{ma}), writing a pointer (C_{mp}) to the hashed tuple, and two pointer writings ($2C_{mp}$) to make connections to other buckets. No searching for duplicate checking is necessary. Hence, the cost of creating $n_f - 1$ indexes on $F_i.A_i$s for $i = 2, 3, \cdots, n_f$, where F_1 is the pivot RF, is computed as follows.

$$C_{ixcrt} = \sum_{i=2}^{n_f}(C_{ma} + C_{colscan}(N_{f_i}) + (C_{hc} + C_{ma} + 3C_{mp})N_{f_i}) \quad (21)$$

Navigational join (Algorithm 3.11): Allocating an empty SNR costs C_{ma}. For the assembly cost (Algorithm 3.12), we consider only the costs of the following operations on data tuples: finding matching tuples (Algorithm 3.13), executing assembly plans (AP) on the found tuples, and inserting (Algorithm 3.8) the resulting tuples into the SNR after duplicate checking (Algorithm 3.6).

Matching (Algorithm 3.13): The cost of Match(t_i, F_j, $t_i.A\ \theta\ t_j.B$), denoted by $C_{match_{ij}}$, is computed as follows. First, hashing a join attribute costs C_{hc}. Let N_b denote the expected length of the bucket chain including the header bucket. Then, in Step 2, it costs $N_b(2C_{mp}+C_e)$ to follow the bucket chain—one C_{mp} for reading a pointer to the tuple $t_j \in F_j$, another C_{mp} for reading a pointer to the next bucket, and C_e for evaluating the join predicate $t_i.A\ \theta\ t_j.B$. α_{ij} tuples of F_j are collected from Match(t_i,F_j,$t_i.A\ \theta\ t_j.B$). Collecting the matching tuples incurs only the cost of writing α_{ij} pointers (i.e., $C_{mp}\alpha_{ij}$). Thus, the cost of finding matching tuples from F_j for all tuples (t_is) of F_i is computed as a function of α_{ij} as follows:

$$C_{match_{ij}}(\alpha_{ij}) = C_{hc} + N_b(2C_{mp} + C_e) + C_{mp}\alpha_{ij} \tag{22}$$

where N_b is obtained as

$$N_b = \mathrm{MAX}\left(N_{f_j}/D_{f_{ij}}, 2\right) \tag{23}$$

$$= \mathrm{MAX}\left(\alpha_{ij}, 2\right) \text{ by Equation 3} \tag{24}$$

in the same way as Equation 10. (As mentioned in Section 3.3.3, we assume the allocated bucket header size is half the hashed RF cardinality.)

The cost of the entire matching process is the sum of the cost of scanning the pivot RF linearly and the cost of finding matching tuples from the other RFs,

$$C_{match} = C_{colscan}(N_{f_1}) + \sum_{i \notin Leaf(JT)} L_{f_i} C_{match_{ij}}(\alpha_{ij}) \tag{25}$$

where Leaf(JT) denotes the set of the JT leaves, and L_{f_i} is obtained as follows:

$$L_{f_i} = N_{f_1} \prod_{\langle RFJT(F_p),\mathrm{RFJT}(F_q)\rangle \in P_{1i}} \alpha_{pq} \tag{26}$$

where P_{1i} is a path from RFJT(F_1) to RFJT(F_i).

Execution of assembly plans. (Step 3*a* of Algorithm 3.12): RF tuples that are found by the matching process are merged as prescribed in the assembly plan. If we let m_i be the number of RF tuples that are merged to produce S_i tuples, and let T'_{s_j},j=1,2, $\cdots$,m_i, denote the size of the attributes projected from each to-be-merged RF, then

$$T_{s_i} = \sum_{j=1}^{m_i} T'_{s_j} \tag{27}$$

Merging two RF tuples requires two projections. Generalizing this case, we obtain the cost of merging m_i RF tuples into one S_i tuple as $\sum_{j=1}^{m_i} C_{project}(T'_{s_j})$. Using Equations 8 and 27, it can be rewritten as a function of T_{s_i} and m_i as follows:

$$C_{apexec_i}(T_{s_i}, m_i) = (m_i - 1)C_{pi} + C_{project}(T_{s_i}) \tag{28}$$

Because n_s nested subrelations are produced out of n_f RFs, $n_f - n_s$ mergings are performed. It depends on a query to determine which RFs are merged to produce each nested subrelation S_i. Let us consider a set of $n_f - 1$ α_{ij}s that are defined among n_f RFs. We define a partition on this set, that is, $[\Gamma_1|\Gamma_2|\cdots|\Gamma_{n_s}]$ where each $\Gamma_k, k=1,2,\cdots,n_s$, is the set of F_is that are merged to produce S_k. Let γ_k denote the combined value of all α_{ij}'s to the F_js in Γ_k and be defined as follows:

$$\gamma_k = \prod_{F_j \in \Gamma_k} \alpha_{ij} \text{ where } \alpha_{i1} = 1 \tag{29}$$

Then, the total cost of executing an assembly plan is computed as follows:

$$C_{apexec} = \sum_{i=1}^{n_s} M_{f_i} C_{apexec_i}(T_{s_i}, m_i) \tag{30}$$

where M_{f_i} is the number of tuples produced for S_i and is computed as follows:

$$M_{f_i} = N_{f_1} \prod_{NSRNFT(S_p) \in P_{1i}} \gamma_p \tag{31}$$

where P_{1i} is the path from NSRNFT(S_1) (i.e., the NFT root) to NSRNFT(S_i).

Searching (Algorithm 3.6) and *Insertion* (Algorithm 3.8): The M_{f_i} tuples are to be inserted into S_i. For each tuple, searching (for duplicate checking) costs $\sum_{i=1}^{n_s} C_{sisearch}(M_{f_i} N_{s_i}, N_{s_{par(i)}})$ and insertion costs $\sum_{i=1}^{n_s} C_{siinsert}(N_{s_i}, T_{s_i}, N_{s_{par(i)}})$.

Thus, the total cost of performing navigational joins on RFs is obtained as follows:

$$\begin{aligned} C_{navjn} &= C_{match} + C_{apexec} + \sum_{i=1}^{n_s} (C_{sisearch}(M_{f_i}, N_{s_i}, N_{s_{par(i)}}) + \\ &\quad C_{siinsert}(N_{s_i}, T_{s_i}, N_{s_{par(i)}})) \end{aligned} \tag{32}$$

SNR Nesting Cost and Assembly Cost.

Nesting: Ignoring the difference between server and client speeds, the only difference between SNR and RF nesting is that tuples produced in the navigational join step are transmitted to a client. The transmission cost is considered separately in Section 4.2.3; therefore the SNR nesting cost is the same as the RF nesting cost.

$$C_{snrnest} = C_{rfnest} \tag{33}$$

Assembly (Algorithm 3.14): There is an additional cost of assembling the received data stream into an SNR on a client. Considering only the cost of operating on tuples (not on the delimiters), the assembly cost is computed as follows:

$$C_{snrassem} = \sum_{i=1}^{n_s} C_{siinsert}(N_{s_i}, T_{s_i}, N_{s_{par(i)}}) \tag{34}$$

4.2.3 Transmission Cost. We use a simple model (Dwyer and Larson, 1987) of data transmission cost defined as follows:

$$\text{Transmission cost} = C_l + C_b \times \text{Size} \tag{35}$$

where Size is the number of bytes of the transmitted data.

In the SFR method, Size is the SFR size $N_t\ T_t$:

$$C_{sfrtx} = C_l + C_b N_t T_t \tag{36}$$

In the RF method, it is the total RF sizes ($N_{f_i}\ T_{f_i}$, $i = 1,2,\cdots, n_f$):

$$C_{rftx} = C_l + C_b \sum_{i=1}^{n_f} N_{f_i} T_{f_i} \tag{37}$$

In the SNR method, it is the total SNR sizes ($N_{s_i}\ T_{s_i}$, $i = 1,2,\cdots, n_s$), ignoring the size of the header and delimiters:

$$C_{snrtx} = C_l + C_b \sum_{i=1}^{n_s} N_{s_i} T_{s_i} \tag{38}$$

5. Cost Comparison

We selected RF data parameters, β_{ij}s, and d_t as a base set, and derived the values of the other data parameters using the formulas shown in Appendix A. We also selected two data parameters—the selectivity (α_{ij}s) and the extra join attribute (EJA) ratio (ρ_{f_i}s)—as the variant parameters. The value of α_{ij} is an indicator of the overhead on an SFR due to duplicate subtuples or on an SNR due to duplicate nested subtuples. Higher selectivities implicate more tuples in an SFR or nested subrelations of an SNR for a given set of RFs. On the other hand, the value of ρ_{f_i} is an indicator of the overhead on RFs due to EJAs. Higher EJA ratios implicate smaller tuples in an SFR or nested subrelations of an SNR for a given set of RFs.

We carry out the cost comparison in two ways: simulation and sample case test. We first show the simulation result obtained using random values of data

parameters. Then, we observe the cost dependency on the variant data parameter values. This observation is reinforced by another round of simulation, this time with biases given to the value ranges of the variant data parameters.

5.1 Overall Comparison Using Simulation

We computed the average costs of the SFR, RF, and SNR methods, and tallied the winning counts—the number of times each method incurred the minimum cost among the three methods. We used a query whose JT and NFT are both a complete binary tree of 7 nodes.[7] (Figure 5). The base data parameter values were randomly selected from the following ranges. (*Ψ denotes* $\{<1,2>,<1,3>,<2,4>,<2,5>,<3,6>,<3,7>\}$. *The numbers tagged with a † are arbitrary "realistic" values. Others are theoretical bounds.*)

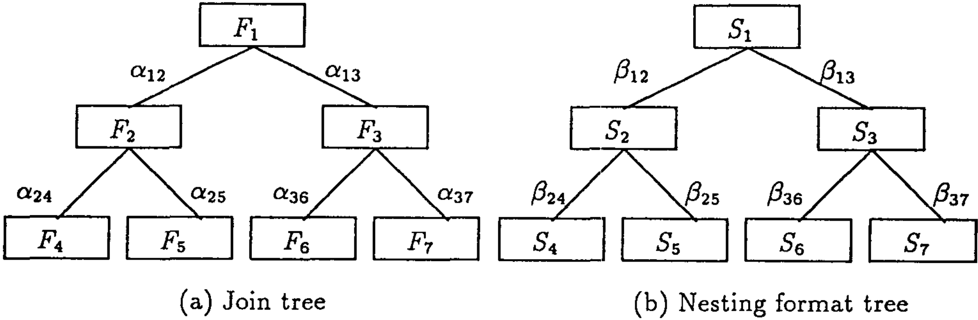
- $10^{\dagger} \leq N_{f_1} \leq 500^{\dagger}$, $10^{\dagger} \leq N_{f_j} \leq N_{f_i}\alpha_{ij}$ for $j = 2,3,\cdots,7$ (satisfying Equation 4).
- $10^{\dagger} \leq T_{f_j} \leq 500^{\dagger}$ for $j = 2,3,\cdots,7$.
- $1.00 \leq \alpha_{ij} \leq 10.00^{\dagger}$ for $<i,j> \in \Psi$.
- $0.50^{\dagger}\alpha_{1j} \leq \beta_{1j} \leq 1.00\ \alpha_{1j}$ for $j = 2,3$ (See Equation 41),
  $0.50^{\dagger}\alpha_{ij} \leq \beta_{ij} \leq 1.50^{\dagger}\alpha_{ij}$ for $<i,j> \in \Psi$ and $i \neq 1$.
- $0.00 < \rho_{f_i} \leq 1.00$ for $i = 1,2,\cdots,7$.
- $0.30^{\dagger} \leq d_{f_j} \leq d_t \leq 1.00$ for $j = 1,2,\cdots,7$.

Some of the value ranges need a justification. First, an α_{ij} value is typically far less than 1 (Christodoulakis, 1984; Valduriez, 1987) for a conventional relational join. In the case of a join between RFs however, it is always ≥ 1 because non-matching tuples have already been discarded in the query materialization step. Secondly, there is a correspondence between α_{ij} and β_{ij} as we can see from the JT and NFT of Figure 5. Their values are not quite similar because nested subrelations in an SNR do not have EJAs and so may have some duplicate tuples eliminated in the nesting step. We picked up β_{ij} values from within $\pm 50\%$ of α_{ij} values, except β_{1j} for which the upper limit is α_{ij} because $N_{s_1} = N_{f_1}$ (Equation 41). Third, $d_{f_j} \leq d_t$ is always true except when the combined domain cardinality (the number of distinct values) of the EJAs is higher than that of the other attributes. (This case is rare.)

---

7. For simplicity we assumed that no RF merging was needed in the nesting step. Its effect on the total cost is negligible. As a result, we used $\gamma_1 = 1$, $\gamma_j = \alpha_{ij}$ for $<i,j> \in \Psi$ and $j \neq 1$, and $m_i = 1$ for $i = 1,2,\cdots,7$ (See Equation 28).

Figure 5. Sample query for simulation



(a) Join tree (b) Nesting format tree

Table 3. Simulation result

Method	Transmission				Partial local processing	
	Average data size	Average cost		#wins	Average cost	#wins
		LAN	WAN			
SFR	3413 Mbytes	3.2 hours	2.4 days	0%	2.9 hours	0%
RF	2.4 Mbytes	8.1 secs	2.4 mins	67%	15.2 secs	100%
SNR	3.2 Mbytes	11.1 secs	3.2 mins	33%	17.5 secs	0%

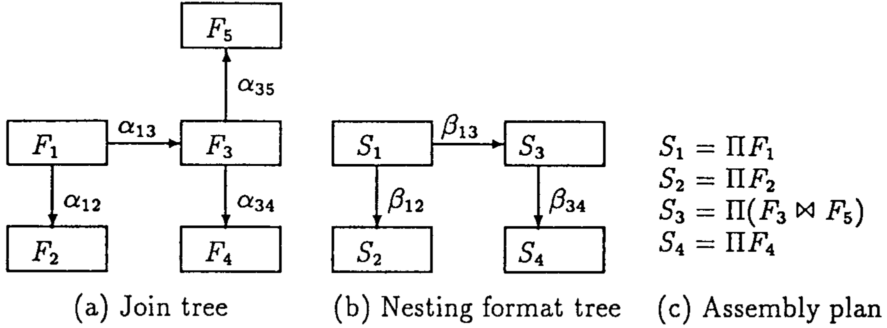
(Transmission time is elapsed time and local processing time is CPU time.)

Table 3 shows the average values and the winning counts (in %) obtained from 5,000 random test cases. The RF and SNR methods showed orders of magnitude improvement compared to the SFR method for both the transmission and local processing costs. The RF method won over the SNR method more frequently, and there was no case where the RF method lost to the SFR method although it could happen in theory. Since we assumed that a server and a client run at the same speed, the SNR method always takes the same cost as the RF method and an additional cost (Equation 34) of assembling an SNR. Therefore, the RF method always shows less local processing cost than the SNR method. The LAN and WAN transmissions showed the same relative cost between any two methods.

5.2 Dependency on Selectivity and Extra Join Attribute Ratio

5.2.1 Observation Using Sample Case Test. We continued cost comparisons using sample values of data parameters and observed the dependency of the costs on the values of a single α_{ij} and a set of ρ_{f_i}, $i = 1,2,\cdots,5$. Figure 6 shows the JT, NFT, and their associated assembly plan of a sample query. Note that F_3 and F_5 are

Figure 6. Sample query for the sample case test



(a) Join tree (b) Nesting format tree (c) Assembly plan

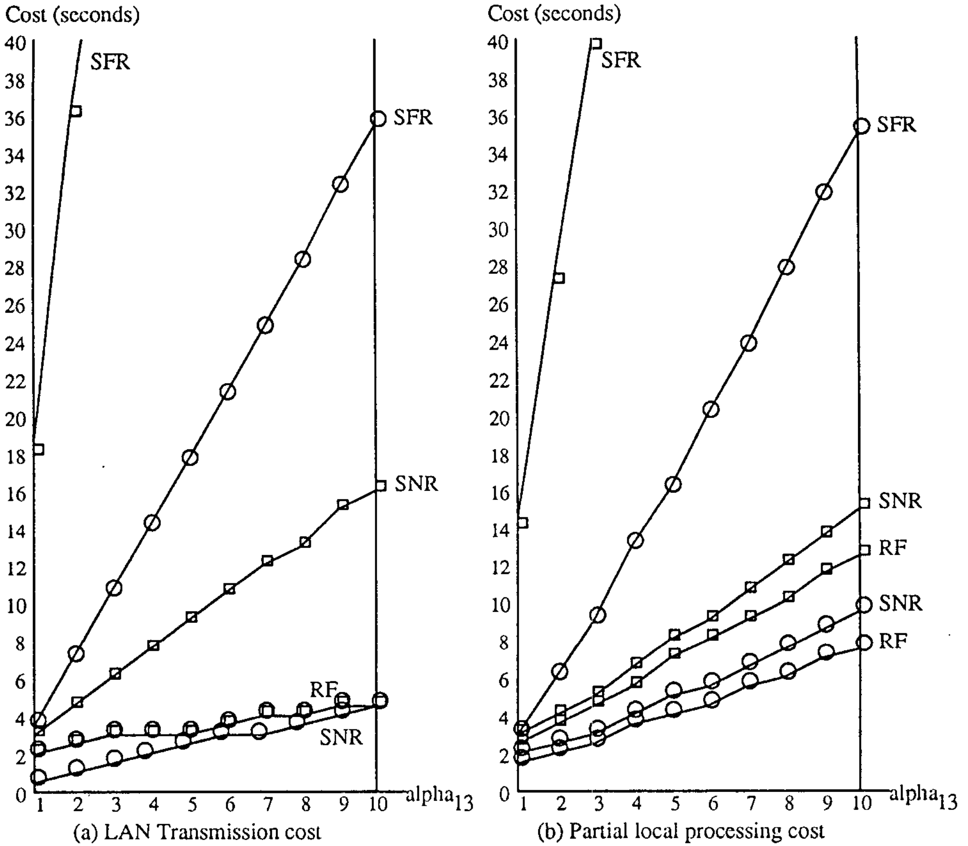
merged (by a join and projection) to produce S_3. The sample values of the base data parameters are as follows:

- N_{f_i} = 500, 800, 300, 1200, 300 for i = 1,2,3,4,5, respectively (satisfying Equation 4).
- T_{f_i} = 200, 300, 250, 100, 400 for i = 1,2,3,4,5, respectively.
- $\alpha_{12} = 3.0$, $\alpha_{13} = 1.0 \sim 10.0$, $\alpha_{34} = 4.0$, $\alpha_{35} = 1.0$. ($\alpha_{35} = 1.0$ because F_3 and F_5 are merged.)
- $\beta_{12} = 2.7$, $\beta_{13} = 0.9\ \alpha_{13}, \beta_{34} = 3.8$ (satisfying $\beta_{1j} \leq \alpha_{1j}$ discussed in Section 5.1).
- $\rho_{f_i} = \begin{cases} \text{0.05,0.1,0.15,0.05,0.05(lower values)} \\ \text{0.8,0.9,0.7,0.6,0.9(higher values)} \end{cases}$ for $i = 1, 2, 3, 4, 5$ respectively.
- $d_t = d_{f_i} = 0.8$ for i = 1,2,3,4,5.

We evaluated the costs while varying the value of α_{13} from 1 through 10. The same evaluation has been repeated for the two sets of ρ_{f_i} values. Figure 7 shows the costs of the three methods with respect to the values of α_{13} and ρ_{f_i}s.

It shows that both the transmission and the local processing costs increase as the value of α_{13} increases, and the slope was in the order of the SFR, SNR, and RF methods from the highest first. Increasing the value of α_{13} without changing the value of $D_{f_{13}}$ is equivalent to increasing the value of N_{f_3} (Equation 3). In the RF method, this increases the size of only F_3 and has no effect on the sizes of the other RFs. On the other hand, it has a "ripple effect" on the size of an SFR or SNR. Increasing N_{f_3} also increases β_{13}, which is amplified by a factor of $N_{s_1}\beta_{12}\beta_{34}$ (Equation 39).

Figure 7. Sample case test result



(The abscissa is the value of α_{13} and the ordinate is the cost in seconds. Lines labeled with boxes or circles are those obtained for lower or higher values of ρ_{f_i}'s, respectively.)

It also shows that costs are smaller for the higher values of ρ_{f_i}s. One exception was the RF transmission cost, in which case the transmission cost is independent of the ρ_{f_i} values (see Equation 37). In particular, the SNR transmission incurred less cost than the RFs for the higher values of ρ_{f_i}s.

5.1.2 Observation Using Simulation. We performed another simulation using the same ranges as in Section 5.1 except for α_{ij}s and ρ_{f_i}s. The following two different ranges were used for these two:

- Range HL: (Higher α_{ij} and lower ρ_{f_i}.)
  $5.00 \leq \alpha_{ij} \leq 10.00$ for $<i,j> \in \Psi$ and $0.00 < \rho_{f_i} \leq 0.50$ for $i = 1,2,\cdots,7$.

- Range LH: (Lower α_{ij} and higher ρ_{f_i}.)
  $1.00 \leq \alpha_{ij} \leq 5.00$ for $<i,j> \in \Psi$ and $0.50 \leq \rho_{f_i} \leq 1.00$ for $i = 1,2,\cdots,7$.

Table 4. Simulation results for biased variant data parameter ranges

Method	Transmission				Partial local processing	
	Average data size	Average cost		#wins	Average cost	#wins
		LAN	WAN			
SFR	33878 Mbytes	32.0 hours	23.5 days	0%	30.2 hours	0%
RF	4.1 Mbytes	14.0 secs	4.1 mins	93%	31.7 secs	100%
SNR	8.8 Mbytes	29.9 secs	8.8 mins	7%	36.6 secs	0%

(a) Range HL ($5.00 \leq \alpha_{ij} \leq 10.00, 0.00 < \rho_{f_i} \leq 0.50$)

Method	Transmission				Partial local processing	
	Average data size	Average cost		#wins	Average cost	#wins
		LAN	WAN			
SFR	47.0 Mbytes	2.7 mins	46.9 mins	0%	2.0 mins	0.8%
RF	0.86 Mbytes	2.9 secs	51.8 secs	22%	4.0 secs	99.2%
SNR	0.53 Mbytes	1.8 secs	31.8 secs	78%	4.6 secs	0%

(b) Range LH ($1.00 \leq \alpha_{ij} \leq 5.00, 0.50 \leq \rho_{f_i} \leq 1.00$)

(Transmission time is elapsed time and local processing time is CPU time.)

Table 4 shows the simulation result. The RF method shows better performance than in Table 3 for Range HL, and worse performance for Range LH. There are even some cases in Range LH where the SFR method is better than the RF method for the partial local processing cost. These results confirm that the observations made in Section 5.2.1 are generally true.

6. Conclusion

We have developed three different methods—SFR and two new methods (RF and SNR)—for instantiating view-objects from a remote relational (preferably main memory) database server by materializing a view query, restructuring the query result into a nested relation, and resolving references among them. Rigorous algorithms have been developed for each step of the methods with a primary focus on the transmission and nesting steps, and a partial cost model has been developed.

Cost comparison results showed that the RF and SNR methods are far more efficient than the SFR method. The RF method wins over the SNR method more

frequently and therefore is the more preferred method. Alternatively, there remains an optimization issue of choosing either the RF or SNR method depending on the query and the speeds of the server and client. The RF and SNR methods are useful in a local database system environment as well, because they perform better, even for the local processing costs alone.

We assumed unlimited main memory for the cost model, which is not always true in a real situation. It may reveal interesting (not opposite) results to elaborate on the cost model by considering the available main memory size in a virtual memory architecture.

References

Abiteboul, S. and Bidoit, N. Non-first normal form relations to represent hierarchically organized data. *Proceedings of the ACM International Conference on the Principles of Database Systems*, Waterloo, 1984.

Abiteboul, S. and Kanellakis, P. Object identity as a query language primitive. *Proceedings of the International ACM SIGMOD Conference on the Management of Data*, Portland, OR, 1989.

Ammann, A., Hanrahan, M., and Krishnamurthy, R. Design of a memory resident DBMS. *Proceedings of the IEEE Computer Conference (COMPCON)*, San Francisco, 1985.

Barsalou, T. View objects for relational databases. Ph.D. thesis, Medical Information Sciences Program, Computer Science Department, Stanford University, 1990.

Barsalou, T. and Wiederhold, G. Complex objects for relational databases. *Computer Aided Design,* (Special issue on object-oriented techniques for CAD), 22(8):458-468, 1990.

Barsalou, T., Siambela, N., Keller, A., and Wiederhold, G. Updating relational databases through object-based views. *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, Denver, CO, 1991.

Barsalou, T., Sujansky, W., and Wiederhold, G. Expert database systems in medicine–The PENGUIN project. *Proceedings of the AAAI Spring Symposium on AI in Medicine*, Stanford University, 1990.

Bitton, D. The effect of large main memory on database systems. *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, Washington, DC, 1986.

Bitton, D. and Turbyfill, C. Performance evaluation of main memory database systems. Technical Report 86-731, Department of Computer Science, Cornell University, 1986.

Bitton, D., Hanrahan, M., and Turbyfill, C. Performance of complex queries in main memory database systems. *Proceedings of the IEEE International Conference on Data Engineering*, Los Angeles, 1987.

Christodoulakis, S. Implications of certain assumptions in database performance evaluation. *ACM Transactions on Database Systems*, 9(2):163-186, 1984.

Codd, E. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377-387, 1970.

DeWitt, D., Katz, R., Olken, F., Shapiro, L., Stonebraker, M., and Wood, D. Implementation techniques for main memory database systems. *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, Boston, MA, 1984.

Dill, G. Peripheral semiconductor storage: A feasible alternative to disk and tape? *Hardcopy*, 7(1):107-113, 1987.

Dittrich, K. and Lorie, R. Object-oriented database concepts for engineering applications. Technical Report RJ 4691 (50029), IBM Research Lab., San Jose, CA, May, 1985.

Dwyer, P. and Larson, J. Some experiences with a distributed database testbed system. *Proceedings of the IEEE*, 75(5):633-648, 1987.

Fischer, P. and Thomas, S. Operators for non-first-normal-form relations. *Proceedings of the IEEE Computer Software and Applications Conference*, Chicago, IL 1983.

Haskin, R. and Lorie, R. On extending the functions of a relational database system. *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, Orlando, FL, 1982.

Horowitz, E. and Sahni, S. *Fundamentals of Data Structures.* London: Computer Science Press, Inc., 1976.

Khoshafian, S. and Copeland, G. Object identity. *Proceedings of the International ACM OOPSLA Conference on Object-Oriented Programming Systems, Languages, and Applications*, Portland, OR, 1986.

Knuth, D. *The Art of Computer Programming,* Vol 3: Sorting and Searching. Reading, MA: Addison-Wesley, 1973.

Law, K., Barsalou, T., and Wiederhold, G. Management of complex structural engineering objects in a relational framework. *Engineering with Computers*, 6:81-92, 1990*a*.

Law, K., Wiederhold, G., Barsalou, T., Siambela, N., Sujansky, W., Zingmond, D., and Singh, H. An architecture for managing design objects in a sharable relational framework. *International Journal of Systems Automation: Research and Applications (SARA)*, 1(1):47-66, 1991.

Law, K., Wiederhold, G., Barsalou, T., Siambela, N., Sujansky, W., and Zingmond, D. Managing design objects in a sharable relational framework. *Proceedings of the ASME International Conference on Computers in Engineering*, Boston, 1990*b*.

Lee, B. Efficiency in Instantiating Objects from Relational Databases through Views, Ph.D. thesis, Computer Science Department, Stanford University, December, 1990.

Lee, B. and Wiederhold, G. Outer joins and filters for instantiating objects from relational databases through views. *IEEE Transactions on Knowledge and Data Engineering,* 6(1):108-119, 1994.

Lehman, T. and Carey, M. A study of index structures for main memory database management systems. *Proceedings of the International Conference on Very Large Data Bases*, Kyoto, Japan, 1986*a*.

Lehman, T. and M. Carey. Query processing in main memory database management systems. *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, Washington, DC, 1986*b*.

Lorie, R. and Plouffe, W. Complex objects and their use in design transactions. *Proceedings of the IEEE Annual Meeting-Database Week: Engineering Design Applications*, 1983.

Mauer, W. and Lewis, T. Hash Table Methods. *ACM Computing Surveys*, 7(1):5-20, 1975.

Roth, M., Korth, H., and Silberschatz, A. Extended algebra and calculus for nested relational databases. *ACM Transactions on Database Systems*, 13(4):389-417, 1988.

Shapiro, L. Join processing in database systems with large main memories. *ACM Transactions on Database Systems*, 11(3):239-264, 1986.

Singh, H. View-objects in CIM environment. Annual Report of the Center for Integrated Systems (CIS), Stanford University, 1990.

Swami, A. Optimization of large join queries. Ph.D. thesis, Computer Science Department, Stanford University, 1989.

Valduriez, P. Join indices. *ACM Transactions on Database Systems*, 12(2):218-246, 1987.

Whang, K., Ammann, A., Bolmarcich, A., Hanrahan, M., Hochgesang, G., Huang, K., Khorasani, A., Krishnamurthy, R., Sockut, G., Sweeney, P. Waddle, V., and Zloof, M. Office-by-example: An integrated office system and database manager. *ACM Transactions on Office Information Systems*, 5(4):393-427, 1987.

Whang, K. and Krishnamurthy, R. Query optimization in a memory-resident domain relational calculus database system. *ACM Transactions on Database Systems*, 15(1):67-95, 1990.

Wiederhold, G. Views, objects, and databases. *IEEE Computer*, 19(12):37-44, 1986.

Wiederhold, G., Barsalou, T., and Sujansky, W. Sharing information among biomedical applications. *Proceedings of the Conference on Software Engineering in Medical Informatics*, Amsterdam, 1990.

Wilkes, W., Klahold, P., and Schlageter, G. Complex and composite objects in CAD/CAM databases. *Proceedings of the IEEE International Conference on Data Engineering*, Los Angeles, 1989.

Appendix

Derivation of Non-base Data Parameters

Provided with the base set of data parameters (Section 5), the other data parameters are derived as follows. Consider N_t as the number of tuples generated when we "flatten" a corresponding SNR. The cardinality of S_1 is N_{s_1}, and each tuple of S_i is replicated β_{ij} times when flattened with its nested subrelation S_j. Hence,

$$N_t = N_{s_1} \prod_{\langle NSRNFT(S_i), NSRNFT(S_j)\rangle \in E(NFT)} \beta_{ij} \tag{39}$$

where $E(\text{NFT})$ denotes the set of edges in the NFT. $\langle$NSRNFT(S_i), NSRNFT(S_j) $\rangle \in E$(NFT) means that S_j is an immediate nested subrelation of S_j.

Since corresponding SFR and SNR have the same set of attributes,

$$T_t = \sum_{k=1}^{n_s} T_{s_k} \tag{40}$$

Since both S_1 and F_1 contain the pivot relation key,

$$N_{s_1} = N_{f_1} \tag{41}$$

and the other N_{s_i}s ($i \neq 1$) are computed from Equation 5 as follows:

$$N_{s_k} = N_{s_1} \prod_{\langle NSRNFT(S_p), NSRNFT(S_q)\rangle \in P_{1k}} \beta_{pq} \text{ for } k = 2, 3, \cdots, n_s \tag{42}$$

where P_{1i} denotes the path from NSRNFT(S_1) to NSRNFT(S_i) in the NFT.

A nested subrelation S_k has no EJAs. Therefore, $T_{s_i} = T_{f_i}(1\text{-}\rho_{f_i})$ if no merging of RFs is needed. In general, T_{s_k} is the total size of RF tuples merged to produce S_k after stripped off their EJAs. (See Section 4.2.2 for Γ_k.)

$$T_{s_k} = \sum_{F_i \in \Gamma_k} T_{f_i}(1 - \rho_{f_i}) \tag{43}$$