Monet; a next-Generation DBMS Kernel For Query-Intensive Applications

Boncz, P.A.

Citation for published version (APA):

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

Disclaimer/Complaints regulations
If you believe that digital publication of certain material infringes any of your rights or (privacy) interests, please let the Library know, stating your reasons. In case of a legitimate complaint, the Library will make the material inaccessible and/or remove it from the website. Please Ask the Library: https://uba.uva.nl/en/contact, or a letter to: Library of the University of Amsterdam, Secretariat, Singel 425, 1012 WP Amsterdam, The Netherlands. You will be contacted as soon as possible.
Chapter 7

Monet as RDBMS back-end

The purpose of this chapter is to illustrate how all functionality of MIL and its implementation in Monet can be used to build a full-fledged DBMS. Recall that in our DBMS architecture of Figure 3.2, this entails the design and implementation of a front-end system, where Monet serves as back-end and the interface between the two is MIL. Rather than describing a completed system, we limit ourselves here to a sketch of such a system, yet go into sufficiently deep detail of its implementation – sometimes ad nauseam – until we get at the level of the exact MIL code needed for providing each query processing functionality.

7.1 Introduction

While Monet can be used to manage object-oriented or even graph-like data models [dVW98, WvZF98], we chose our example case here in the relational field, as this is most common and therefore provides ample reference to other systems. Specifically, we discuss a front-end that provides basic SQL-2 compliant RDBMS functionality. Again, while Monet has been shown to be able to store and query multi-media (audio, video), XML and GIS data types [BQK96, NK97, SKWW00], we limit ourselves here to common business data types. Making the case even more specific, we use a particular database schema and filling in our examples, namely the TPC-H (formerly known as TPC-D) database, as this well-defined DBMS benchmark is targeted to the focal point of the Monet design: analytical query-intensive environments (OLAP, Data Mining).

Functionally, our SQL-2 RDBMS, consisting of Monet as back-end, and the front-end system discussed here, should:

- provide high transaction processing (TP) rates on operational tables of moderate size (such that these tables fit in memory).
- provide high analytical performance (OLAP, Data Mining) on voluminous historical (“data warehousing”) data tables.
- provide a mechanism of fine-grained update propagation between the quickly changing operational tables, and the larger and stabler data warehousing images.

1Check monetdb.cwi.nl for the ongoing efforts of our group.
The motivation to choose this example setting is twofold. First, it is a showcase to demonstrate not only how analytical queries are processed by a RDBMS based on Monet, but it also shows how updates and transactions can be handled with MIL.

Second, there is strong market relevance for this particular setting: DBMS technology that combines high performance in both transactions and analytical queries. Current DBMS products can often only deliver one of these capabilities at a time. This is one of the reasons why many organizations employ separate OLTP and data warehousing DBMS environments. Such organizations have built data warehousing systems that are filled periodically by copy management tools that periodically extract data from operational systems. Such operational systems are typically highly transaction-oriented, sustain a heavy load of simple update and read requests, and contain the freshest image of the reality that the organization records in its IT infrastructure. The data warehousing system on the other hand, typically deals with data that is somewhat stale, and is optimized to sustain analytical read-mostly loads consisting of complex queries, that are generated by reporting, OLAP and Data Mining tools. Data volumes in the data warehouse are often much larger than those found in the operational systems, because operational systems tend to record the current state of affairs, whereas data warehouses typically record historical data, very roughly speaking consisting of a (time-)series of snapshots from the operational systems.

While there are many other reasons (e.g. historical, organizational) to have separate OLTP and data warehousing environments [Imm92, Wid95, CD97], one of them certainly is related to performance problems that RDBMS technology has with sustaining both complex query and OLTP loads at the same time. However, we envision that in the near future, organizations will be inclined to integrate the now separate operational and data warehousing environments again. In commercial practice, one can already observe that new applications are often deployed on top of the data warehouse instead of the OLTP systems, because the data model in the data warehouse is richer (it may unite data from various sources, and contains historical information). However, as these new applications also demand (some) fresh data, organizations find themselves inclined to increase the frequency of propagating updates from the OLTP environments to the data warehouse (e.g. from each month to each night, to each hour, etc.). Obtaining high update frequencies might be difficult to achieve performance-wise. Hence the need for a DBMS that can sustain both high-performance TP and analytical query loads.

Additionally, one can identify emerging DBMS applications that explicitly require this combination:

- e-commerce sites that log visitor information to a RDBMS and want to personalize their web pages (e.g. using profiles found by data mining) to the web browsing actions a visitor performs in real-time. This yields a high query and update load to the tables holding the web log.

- telecommunications companies that want to deploy sophisticated billing policies in real-time on their Call Detail Record (CDR) databases. Again, these policies lead to continuous querying to the CDR tables, which are also continuously being updated.

- financial broker risk analysis, that requires highly frequent recalculation of the

---

2Note our assumption that the transaction database fits in main memory.
risk a financial portfolio poses against a highly active database of financial (stock) price information. Up till now, such real-time risk analysis has only been possible on simplified representations of the real data.

The TPC-H Benchmark

We use the data model of the TPC-H benchmark and some of its queries as our running example throughout this chapter, as its schema and queries are well-known and it fits nicely to the query-intensive DBMS area for which Monet was designed (decision support on large data warehouses).

The TPC-D benchmark started out as the generally accepted yardstick for query-intensive loads on a synthetically generated data warehousing environment. However, a “benchmark war” broke out among the commercial RDBMS vendors that initially had helped specify the TPC-D. The center of this dispute was the question which optimization techniques were allowed in TPC-D implementations. In particular, after having implemented materialized views in their RDBMS, Oracle was able to remove all costly joins from the TPC-D query execution, which improved their results by more than a factor 100\(^3\). The outcome of the conflict that followed was that TPC-D got abolished and two new benchmarks were defined: TPC-R (reporting) and TPC-H (ad-hoc querying). The TPC-H benchmark now represents what TPC-D used to represent: decision support environments where users don’t know which queries will be executed against a database system; hence, it measures ad-hoc query execution performance. Given this ad-hoc-ness, no prior knowledge of the queries can be built into the physical design of the DBMS.

Moreover, in November 1998, Oracle went on to defy its main competitor Microsoft with a “Million Dollar Challenge”: $1 million to anyone who could “demonstrate that SQL Server 7.0 is not 100 times slower than the Oracle database when running a standard business query against a large database.” Needless to say, Microsoft SQL Server by that time did not implement materialized views.

\[\text{Figure 7.1: TPC-D/H/R Schema}\]
Figure 7.1 shows the standard schema used in the TPC-H. This schema models an internationally operating company that buys parts from manufacturers and sells them to customers. Customers place orders, that consist of a number of line-items. Each line-item is a certain quantity of some part. Parts are obtained from manufacturers. The main tables in terms of size are the ORDERS and LINEITEMS, which contain 1.5 and 6 million rows respectively. All tables except NATION and REGION can be scaled with a scaling factor (SF); hence with SF=100 the ORDERS and LINEITEMS table contain 150 and 600 million rows respectively. Currently popular scaling-factors among DBMS vendors are SF=30, SF=100 and SF=300. Figure 7.2 shows some current results. These are benchmarks of commonly known RDBMS products, performed on (mostly) SMP machines with large memories and multiple CPUs. The “QpH” (Query-Per-Hour) metric is a combination of the performance on all 22 analysis queries and 2 update (“data refresh”) batches.

7.1.1 Road Map

The description of our hypothetical SQL-2 RDBMS front-end is not intended to be complete as we bypass many important issues that are relevant in RDBMS (front-end) design, like query transformation and query optimization. Our description is merely intended to illustrate how MIL and Monet can be used to construct a full-fledged DBMS. Hence, we concentrate on how the RDBMS front-end uses Monet and MIL to fulfill its primary tasks. This involves various aspects of DBMS functionality, discussed in the following sections:

physical data storage In Section 7.2, we discuss how a relational schema can be stored in Monet. In this discussion, we use the schema from the TPC-H benchmark to illustrate how this works in detail, also covering the use of auxiliary indexing structures.

query execution Section 7.3 describes how SELECT-FROM-WHERE queries are executed in Monet, by showing how all major SQL language constructs can be
mapped into the MIL language. We also discuss more complex cases like e.g. multi-column joins. A detailed run of Query-9 from the TPC-H benchmark is used to illustrate how MIL achieves efficient execution of OLAP queries.

**transaction management** We show in Section 7.4 how Monet can provide a mechanism for handling update queries. Here we demonstrate that while the individual MIL primitives do not have ACID properties, they provide the sufficient functionality to build a fully ACID and high performance SQL update service.

### 7.2 Physical Storage of Relational Tables

In this section we illustrate how relational table data gets stored in Monet. Figure 7.3 lists an object-oriented data dictionary, where a database consists of a number of tables, that each have a number of columns. Inverted lists may be attached to columns, and join indices may be defined between tables.

This object-oriented structure reflects the internal data structures of the front-end; and might be mapped on BATs so as to store this meta-information in Monet itself.

In the following, we will describe the various kinds of BATs referenced by this data dictionary (i.e. the C-, S-, I-, D- and M-BATs) which are used for all mass-storage, and discuss their role in query processing and the physical resources they claim.

As a concrete example, we study how the TPC-H schema is stored in Monet, search accelerator structures inclusive.

#### 7.2.1 Basic Table Storage in BATs

Relational tables can be stored in Monet using the basic technique described Section 4.3.2, which simply decomposes each table by column and puts each column of type X in a BAT\[void,X\].

Relevant for Monet is implementation rule number 1.5.6 of TPC-H, which states that vertical fragmentation as a physical design strategy is explicitly forbidden. This rule prevents DBMS vendors from placing table columns that are not accessed by the benchmark queries in a separate fragment; which effectively would make the tables smaller (one could call this opportunistic vertical fragmentation). We argue, though, that full vertical fragmentation as employed in Monet, where each column is stored in a separate BAT, should be an exception to this rule, since full vertical fragmentation is not an opportunistic strategy. Queries benefit from full vertical fragmentation because only data that is actually needed is accessed; but also pay a price in terms of extra query effort introduced by fragmentation (data that is needed by the query is also fragmented and needs to be joined together). Key issue is that the vertical fragmentation is not applied to match the specific access pattern of the benchmark queries, but uniformly across the whole schema. As such, full vertical fragmentation is a physical design strategy that uses no pre-knowledge about the query load, and therefore – in our view – does not violate the faith of the TPC-H benchmark.

The purpose of Figure 7.4 is to provide insight into the storage requirements imposed by the decomposition into BATs. For each column of the TPC-H tables, it shows its column name, type as specified by the TPC-H benchmark rules, the corresponding Monet type, the storage requirements in both the “C-BAT” and “S-BAT” representations, and a detailed description of what values are stored in the columns (taken from
Figure 7.3: Data Dictionary Structure in the SQL Frontend
### 7.2. Physical Storage of Relational Tables

<table>
<thead>
<tr>
<th>Column</th>
<th>Type</th>
<th>C-BAT</th>
<th>S-BAT</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>supplkey</td>
<td>oid</td>
<td>4</td>
<td>8</td>
<td>unique from [1..SF*10000]</td>
</tr>
<tr>
<td>name</td>
<td>str</td>
<td>20</td>
<td>24</td>
<td>&quot;SupplierXXXXXX&quot;</td>
</tr>
<tr>
<td>address</td>
<td>str</td>
<td>32</td>
<td>36</td>
<td>&quot;random string [26]&quot;</td>
</tr>
<tr>
<td>nationkey</td>
<td>str</td>
<td>4</td>
<td>8</td>
<td>random value [0..24]</td>
</tr>
<tr>
<td>phone</td>
<td>str</td>
<td>20</td>
<td>24</td>
<td>&quot;IL-RRR-PPP-LLL&quot;</td>
</tr>
<tr>
<td>account</td>
<td>dbl</td>
<td>8</td>
<td>12</td>
<td>&quot;random value [-999.99..9,999.99]&quot;</td>
</tr>
<tr>
<td>comment</td>
<td>str</td>
<td>68</td>
<td>72</td>
<td>&quot;random string, mean length 63&quot;</td>
</tr>
<tr>
<td>supplier</td>
<td></td>
<td>156</td>
<td>184</td>
<td>SF*10,000 rows</td>
</tr>
<tr>
<td>partykey</td>
<td>oid</td>
<td>4</td>
<td>8</td>
<td>unique from [1..SF*200000]</td>
</tr>
<tr>
<td>name</td>
<td>str</td>
<td>40</td>
<td>44</td>
<td>&quot;random string [35]&quot;</td>
</tr>
<tr>
<td>address</td>
<td>str</td>
<td>20</td>
<td>24</td>
<td>&quot;manufacturerX&quot;</td>
</tr>
<tr>
<td>brand</td>
<td>str</td>
<td>12</td>
<td>16</td>
<td>&quot;BrandXX&quot;</td>
</tr>
<tr>
<td>type</td>
<td>enum1[str]</td>
<td>1</td>
<td>8</td>
<td>random string from 150 Types</td>
</tr>
<tr>
<td>size</td>
<td>enum1[int]</td>
<td>1</td>
<td>8</td>
<td>&quot;random value [1..50]&quot;</td>
</tr>
<tr>
<td>container</td>
<td>enum1[str]</td>
<td>1</td>
<td>8</td>
<td>random string from 40 Containers</td>
</tr>
<tr>
<td>retailprice</td>
<td>dbl</td>
<td>8</td>
<td>16</td>
<td>&quot;random string from [9000..00..19999.99]&quot;</td>
</tr>
<tr>
<td>comment</td>
<td>str</td>
<td>28</td>
<td>32</td>
<td>&quot;random string, mean length 14&quot;</td>
</tr>
<tr>
<td>partsupp</td>
<td></td>
<td>115</td>
<td>174</td>
<td>SF*200,000 rows</td>
</tr>
<tr>
<td>custkey</td>
<td>oid</td>
<td>4</td>
<td>8</td>
<td>unique from [1..SF*150000]</td>
</tr>
<tr>
<td>address</td>
<td>str</td>
<td>20</td>
<td>24</td>
<td>&quot;CustomerXXXXXXX&quot;</td>
</tr>
<tr>
<td>nationkey</td>
<td>str</td>
<td>32</td>
<td>36</td>
<td>&quot;random string [25]&quot;</td>
</tr>
<tr>
<td>phone</td>
<td>str</td>
<td>20</td>
<td>24</td>
<td>&quot;IL-RRR-PPP-LLL&quot;</td>
</tr>
<tr>
<td>account</td>
<td>dbl</td>
<td>8</td>
<td>16</td>
<td>&quot;random string [-999.99..9,999.99]&quot;</td>
</tr>
<tr>
<td>comment</td>
<td>str</td>
<td>128</td>
<td>132</td>
<td>&quot;random string, mean length 124&quot;</td>
</tr>
<tr>
<td>customer</td>
<td></td>
<td>166</td>
<td>208</td>
<td>SF*150,000 rows</td>
</tr>
<tr>
<td>orderkey</td>
<td>oid</td>
<td>4</td>
<td>8</td>
<td>unique from [1..SF*6000000]</td>
</tr>
<tr>
<td>orderstatus</td>
<td>chr</td>
<td>1</td>
<td>8</td>
<td>&quot;F&quot;, &quot;O&quot; or &quot;P&quot;</td>
</tr>
<tr>
<td>totalprice</td>
<td>dbl</td>
<td>8</td>
<td>16</td>
<td>random from [8100..7559996.22]</td>
</tr>
<tr>
<td>orderdate</td>
<td>enum2[date]</td>
<td>2</td>
<td>8</td>
<td>random from [start..end-151]</td>
</tr>
<tr>
<td>orderpriority</td>
<td>enum1[str]</td>
<td>1</td>
<td>8</td>
<td>random string from 5 Priorities</td>
</tr>
<tr>
<td>clerk</td>
<td>str</td>
<td>16</td>
<td>20</td>
<td>&quot;ClerkXXXXXX&quot;</td>
</tr>
<tr>
<td>shippriority</td>
<td>enum1[int]</td>
<td>1</td>
<td>8</td>
<td>always 0</td>
</tr>
<tr>
<td>comment</td>
<td>str</td>
<td>56</td>
<td>60</td>
<td>&quot;random string, mean length 49&quot;</td>
</tr>
<tr>
<td>order</td>
<td></td>
<td>93</td>
<td>144</td>
<td>SF*1,500,000 rows</td>
</tr>
<tr>
<td>linenum</td>
<td>oid</td>
<td>4</td>
<td>8</td>
<td>random from [1..SF*6000000]</td>
</tr>
<tr>
<td>quantity</td>
<td>enum1[int]</td>
<td>1</td>
<td>8</td>
<td>unique within order from [1..7]</td>
</tr>
<tr>
<td>extendedprice</td>
<td>dbl</td>
<td>8</td>
<td>16</td>
<td>random from [9000..00..99999.99]</td>
</tr>
<tr>
<td>discount</td>
<td>enum1[dbl]</td>
<td>1</td>
<td>8</td>
<td>random from [0.00..0.10]</td>
</tr>
<tr>
<td>tax</td>
<td>enum1[dbl]</td>
<td>1</td>
<td>8</td>
<td>random from [0.00..0.08]</td>
</tr>
<tr>
<td>returnflag</td>
<td>chr</td>
<td>1</td>
<td>8</td>
<td>&quot;R&quot;, &quot;A&quot; or &quot;N&quot;</td>
</tr>
<tr>
<td>linestatus</td>
<td>chr</td>
<td>1</td>
<td>8</td>
<td>&quot;O&quot; or &quot;F&quot;</td>
</tr>
<tr>
<td>shiptode</td>
<td>enum2[date]</td>
<td>2</td>
<td>8</td>
<td>random from [start+1..end-30]</td>
</tr>
<tr>
<td>committode</td>
<td>date</td>
<td>2</td>
<td>8</td>
<td>random from [start+30..end-61]</td>
</tr>
<tr>
<td>receiptdate</td>
<td>enum2[date]</td>
<td>2</td>
<td>8</td>
<td>random from [start+1..end-91]</td>
</tr>
<tr>
<td>shhipinstruct</td>
<td>enum1[str]</td>
<td>1</td>
<td>8</td>
<td>random from 4 Instructions</td>
</tr>
<tr>
<td>shhipmode</td>
<td>enum1[str]</td>
<td>1</td>
<td>8</td>
<td>random from 7 Motes</td>
</tr>
<tr>
<td>comment</td>
<td>str</td>
<td>32</td>
<td>36</td>
<td>&quot;random string, mean length 27&quot;</td>
</tr>
<tr>
<td>item</td>
<td></td>
<td>66</td>
<td>104</td>
<td>SF*5,000,000 rows</td>
</tr>
<tr>
<td>nationkey</td>
<td>oid</td>
<td>4</td>
<td>8</td>
<td>unique from [0..24]</td>
</tr>
<tr>
<td>name</td>
<td>str</td>
<td>12</td>
<td>12</td>
<td>&quot;unique string, mean length 7&quot;</td>
</tr>
<tr>
<td>regionkey</td>
<td>oid</td>
<td>4</td>
<td>8</td>
<td>random from [0..4]</td>
</tr>
<tr>
<td>comment</td>
<td>str</td>
<td>100</td>
<td>104</td>
<td>&quot;random string, mean length 95&quot;</td>
</tr>
<tr>
<td>nation</td>
<td></td>
<td>120</td>
<td>122</td>
<td>25 rows</td>
</tr>
<tr>
<td>regionkey</td>
<td>oid</td>
<td>4</td>
<td>8</td>
<td>unique from [0..4]</td>
</tr>
<tr>
<td>name</td>
<td>str</td>
<td>12</td>
<td>16</td>
<td>&quot;unique string, mean length 6&quot;</td>
</tr>
<tr>
<td>region</td>
<td></td>
<td>116</td>
<td>125</td>
<td>9 rows</td>
</tr>
</tbody>
</table>

Figure 7.4: Details of TPC-H storage in BATs
the TPC-H definition). For now, we focus on the representation in C-BATs, which simply stands for “Column BATs”. Storage in S-BATs is discussed in Section 7.2.3 and later. For each TPC-H column described in Figure 7.4 we get a C-BAT named here for convenience “C.<TABLE>_<COLUMN>”, e.g. for the first column we get a BAT[void, oid] named C_SUPPLIER_SUPPKEY.

Each column has a C-BAT (see Figure 7.3), thus we get 61 C-BATs for the 61 columns of the 8 tables in the TPC-H schema. All C-BATs have a void head type, indicating a sequence of densely ascending oid-s that start at zero (0,1,2,3,...). As explained in Section 5.2 this has two advantages: void-s do not take up any space at all, and lookup into void columns is positional and therefore highly efficient.

The choice of what tail type to use in each C-BAT is mainly dictated by the requirements the TPC-H rules imposes on column types. An “identifier” must be able to contain values between minus 2 billion and plus 2 billion, hence require at least a 32-bit integer type (the int type in Monet). Similarly detailed requirements hold for the “decimal” and “date” types, hence those must be represented in Monet as dbl and date (the latter is a user-defined type from the Monet extension module “time”, which uses 32-bits integers to store dates as the (possibly negative) number of days since January 1 of the year 1⁴. This date type is implemented as an int (see Section 5.2.1, “Implementation Types”), hence MIL-selects and joins on dates are executed by the fast int-optimized implementation routines.

Calculating the byte width per tuple in Monet is simple: a char or bit value takes one byte, whereas int and date values take four, and a dbl takes 8. Text values of type str are variable size, hence are stored in Monet in a separate heap for variable-sized data items. The storage per value is one 4-byte integer byte offset in the BAT plus the space taken up in this extra heap. In this heap, string values are stored as standard UTF-8 strings (byte sequences, terminated by a zero byte). Storage in the heap is aligned to 4-byte addresses, hence one should round up the size of the string (zero-terminator inclusive) to the nearest larger-or-equal multiple of four. To complicate this calculation some more, the string type in Monet eliminates double occurrences in a BAT dynamically up to a certain threshold number of different values. That is, on a string column that contains only the strings “male” and “female”, the byte-offsets for all values point in the variable-size heap to the same two values. Hence, on large columns with less than a couple of thousand different values, storage requirements per tuple are just the 4-byte integer offset (as the limited heap-size is amortized among all tuples).

Many columns in the TPC-H tables actually store a sub-range of the domain provided by the required column type. For example LINEITEM.TAX only contains one of the nine values { 0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8} while the implementation rules require to store this column in an 8-byte wide dbl. For such columns, we use the “Enumeration Type” functionality of Monet, described in Section 5.2.1. From the outside, the enum[dbl] tail column in the C LINEITEM.TAX BAT behaves as a normal dbl column. Internally, it is stored in a enum[1] tail, which takes just 1 byte per tuple, and also enables a number of processing optimizations (like “Enumeration Views”, described in Section 5.3.2). In the entire TPC-H schema, we use 13 enum[1] BATs, and four enum[2] BATs, which, for example, reduce the total storage requirements for the ORDERS and LINEITEMS tables, from 112 to 94 bytes and from 94 to 67 bytes, respectively. These byte-per-tuple storage requirements for each C-BAT are listed in

⁴Give or take a couple of months stashed away by various popes in the middle ages.
7.2. PHYSICAL STORAGE OF RELATIONAL TABLES

the fourth column of Figure 7.4. In the title row per table (the bottom row) of this
same fourth column, the total required storage for the table in C-BAT representation
is listed. These are obtained by adding the bytes-per-tuple storage requirements for
all C-BATs of the table, multiplying by the number of rows in the table (which is
parametrized by the TPC-H scaling factor SF), and dividing by $2^{20}$ to get the total
storage required for the table in MB. For example, the SUPPLIERS table takes SF*1.5MB
(e.g., with SF=1 it takes 1.5MB, with SF=100 it takes 150MB, etc.).

Adding all per-table storage requirements, we see that the C-BAT representation
of the TPC-H database in Monet takes SF*0.67GB. This is considerably less than the
projected dataset size of the TPC-H benchmark, which is SF*1.0 GB, and certainly
less than the storage sizes reported on RDBMS implementations of the TPC-H, which
vary between SF*6.9 GB and SF*19.1 GB (see Figure 7.2). This significant difference
is mainly caused by the fact that as a BAT[void,X] stores just values of one type,
storage overheads are reduced to a minimum. Relational tuples have to take into
account alignments (which forces byte-padding) and typically reserve some space in
order to accommodate updates (in case of VARCHAR values, for example). In Monet,
the string storage itself is decomposed into the variable-size atom heap, which does
not need to contain any slack-space, and in addition, can perform double-elimination.

It should be said, that the relational storage “bloat” factors of 7-19 observed in
the last column of Figure 7.2 also (and especially) includes storage space taken up
by indexing structures. Later on, we therefore pay some attention to what index
structures might be useful in Monet for the TPC-H database and what additional
storage this may take.

7.2.2 Updates in Void Columns

The choice of using the void-type in the head of the C-BATs draws complications
when updates are done. Monet allows void-columns to be appended upon only, as the
void-ness of the column implies that it always contains a densely ascending range of
oid-s (0,1,2,3,... etc.). This is not a problem when inserting a new tuple, e.g:

insert into SUPPLIER (SUPPKEY, NAME, ADDRESS, NATIONKEY)
values (666, 'Microsoft', 'One Microsoft Way, Redmond (WA)', 17, '01-425-8828080')

would become in MIL:

C_SUPPLIER_SUPPKEY.insert(37, 666);
C_SUPPLIER_NAME.insert(37, "Microsoft");
C_SUPPLIER_ADDRESS.insert(37, "One Microsoft Way, Redmond (WA)");
C_SUPPLIER_NATIONKEY.insert(37, 17);
C_SUPPLIER_PHONE.insert(37, str(nil));
C_SUPPLIER_ACCTBAL.insert(37, dbl(nil));
C_SUPPLIERCOMMENT.insert(37, str(nil));

The above supposes that all C_SUPPLIER.* BATs have densely ascending oid values in
the head columns that end at 36, hence inserting a BUN with value 37 is allowed, and
will keep the head column of these BATs void. Notice that we keep all BATs of equal
length by inserting nil-s if values are missing.

Because oid-s stored in head columns are system-generated, and system-managed,
the RDBMS user will never see them. Therefore, an SQL update will never change
these oid-s, updates just change the values stored in the tail columns:

update SUPPLIER set PHONE = '01-425-8828080' where NAME = 'Microsoft';
Becomes in MIL:

\[
\begin{align*}
&C._{\text{SUPPLIER PHONE}}.\text{replace}(
\quad C._{\text{SUPPLIER NAME}}.\text{select}(\text{"Microsoft"}).\text{project}(\text{"01-425-8828080"});
\end{align*}
\]

The \texttt{replace} uses the \texttt{BAT[oid,str]} that resulted from putting the telephone number string in the tail of all BUNs returned by the \texttt{select("Microsoft")}. Because the \texttt{C._{SUPPLIER PHONE}} has a \texttt{void} head column, the \texttt{replace} can use positional lookup for finding [37,nil], and changing it into [37,\texttt{"01-425-8828080"}].

In the \texttt{select}, we look up the head column \texttt{oid} value for the Microsoft tuple in the \texttt{NAME} column-BAT. Depending on the presence of a hash-table accelerator structure, the \texttt{MIL} implementation of \texttt{select} will use hash-lookup or sequential scan (see Section 5.3.4).\footnote{On "small" \texttt{BAT}s, that can easily be held in memory, \texttt{Monet} automatically creates a hash-table accelerator on such lookup requests.} Since the \texttt{C._{SUPPLIER NAME}} is a \texttt{BAT[void,str]}, the relation is sorted on head column, not on tail column, hence the binary search implementation of \texttt{select} cannot be used. In the discussion of inverted lists (later on) we will discuss how binary search can be used anyway to accelerate such lookup on huge non-memory resident \texttt{BAT}s.

Let us now discuss tuple deletes. How can we maintain a densely ascending \texttt{oid} sequence when a tuple from the middle is deleted? The answer is that we cannot, and we must leave a hole (a dummy element) in all \texttt{C._{SUPPLIER.*}} \texttt{BAT}s. In order to keep track of these holes, we maintain for each relational table an additional "holes" \texttt{BAT[oid,void]} named \texttt{H.<TABLE>} (see also Figure 7.3). This \texttt{H-BAT} stores a list of head \texttt{oid}s that have been deleted and whose positions are effectively free in the \texttt{C._{SUPPLIER.*}} \texttt{BAT}s.

Concerning what to do with the tail values of these "hole" \texttt{BUN}s, two possibilities arise:

- leave them as is. This approach minimizes \texttt{BUN} delete cost. During query processing, we will have to filter out hole-tuples that have been included in a selection or join predicate explicitly.

- replace all tail values with \texttt{nil}. As we will see later, this can later benefit query processing performance, as \texttt{nil}s are automatically excluded from (equi-) joins and most selection predicates.

Both strategies can also be mixed; i.e. those columns that are most often used in selection and join conditions could be nullified, and the others not:

\[
\text{delete from \texttt{SUPPLIER where NAME = 'Microsoft'}}
\]

would become:

\[
\begin{align*}
&\{ \text{var del = C._{SUPPLIER NAME}.select("Microsoft").mark(nil);}
&H._{SUPPLIER}.insert(del); \\
&C._{SUPPLIER SUPPKEY}.replace(del); \\
&C._{SUPPLIER NATIONKEY}.replace(del); \\
&C._{SUPPLIER NAME}.replace(del.project(str(nil))); \\
&C._{SUPPLIER ACCTBAL}.replace(del.project(dbl(nil))); \}
\end{align*}
\]

The first statement in this sequence looks up the \texttt{oid}s of any Microsoft tuples, and puts them in the head column of a \texttt{BAT} with \texttt{nil-oid}s in the tail. The second statement...
7.2. PHYSICAL STORAGE OF RELATIONAL TABLES

registers these oid-s in the hole list for the supplier table. The rest of the statements enforce the nullify policy on the SUPPKEY, NAME, NATIONKEY and ACCTBAL columns. Consequently, in this example the PHONE, ADDRESS and COMMENT follow the leave-as-is deletion policy.

While this scheme allows us to implement relational updates efficiently, and to take full advantage of mass column storage in BATs with void head columns (that take minimal space and offer the possibility of positional lookup of oid values), its downside is that it complicates query execution. Queries get more complicated, because each time a base relation is accessed, holes need to be filtered out explicitly. This filtering, however, can be integrated into normal query processing by including the hole-filter-condition as an implicit selection on each table involved. The general scheme is what corresponds to the leave-as-is deletion policy and is demonstrated as follows:

select NAME from SUPPLIER where PHONE = '01-425-8828080'

should -- if holes would not be an issue -- be translated into MIL:

select(C_SUPPLIER_PHONE, "01-425-8828080").mirror.join(C_SUPPLIER_NAME).print;

However, due to the leave-as-is policy in C_SUPPLIER_PHONE, the Microsoft tuple would now still be found. Holes need to be filtered out, as follows:

select(C_SUPPLIER_PHONE, "01-425-8828080").
mark(nil).diff(H_SUPPLIER).mirror.join(C_SUPPLIER_NAME).print;

The cost for this filtering is not large: the mark has a view implementation and is free in MIL (see Section 5.3.4), and the size of the H_SUPPLIER is small, hence the diff uses an efficient hash-based implementation (making sequential pass over the left operand, and performing hash-lookup into its right operand).

This filtering can often be omitted when columns are accessed on which the nil-replacement deletion policy is used:

select PHONE from SUPPLIER where NAME = 'Microsoft'

becomes in MIL:

select(C_SUPPLIER_NAME, "Microsoft").mirror.join(C_SUPPLIER_PHONE).print;

as the value "Microsoft" had already been replaced by nil during the tuple delete, the MIL select will not find the Microsoft tuple anyway, hence it need not be filtered out later. This optimization works for almost all selection predicates (except isnil(any):bit) and also for joins.

7.2.3 Column Indexing with Inverted Lists

Storage in separate vertical fragments are called in literature “transposed files” [Bat79] or “projection indices” [O’N87], hence one could already consider the BAT storage scheme described above as an “indexing” scheme.

Still, the basic storage in C-BAT(void,X)-s normally stores the tail column unordered, thus if columns are accessed for a SELECT or JOIN query, the MIL select operator has to use a sequential-scan implementation (see Section 5.3.4). For queries that select only a small percentage of the tuples, this is not an optimal solution. One possible improvement would be to create (and maintain) a hash table on each column. The
built-in hash table structure of Monet implements efficient direct hashing and reduces the complexity of the selection operator to $O(sN)$, where $s$ is the selectivity and $N$ the cardinality. This solution is optimal for in-memory situations. The hash-selection algorithm, however, exhibits a random access pattern to the BAT and the hash table structure. If the relational table is so large that all C-BATs together do not fit into main memory, the usual thing to do in Monet is to memory map the disk images of the C-BATs into virtual memory. In such a situation, the hash selection algorithm would cause a significant amount of page faults due to its random access property. Performance can then be improved by ordering the column BATs on tail, so the `select` could use a binary-search algorithm to get to the first matching tuple, and retrieve the other matching tuples in sequential order. Also, the binary search approach on a sorted BAT can also accelerate range-selection, which is impossible with a hash table.

**Physical Storage of Inverted Lists**
 Ordering each column on the tail column would mean that the oid sequence in the head columns would be different for all BATs. As a consequence, the head columns would no longer be stored in void columns, but as oid-s. This both has the disadvantage that storage in BAT[oid,X] takes generally twice the space as storage in BAT[void,T], and secondly it would become much more expensive to look up values by oid. Lookup in an oid column uses a scan, binary-search, or hash-lookup, which – although each is implemented efficiently in Monet – is never as quick as positional lookup in a void-column. Therefore, table storage where each column becomes a BAT[oid,T] ordered on tail is not a good idea.

The idea of ordering on tail becomes only attractive if it is used in addition to the normal decomposition in BAT[void,T]. Hence, we can maintain BAT[oid,T] copies of these BATs that are ordered on tail, which effectively provides us with inverted list search accelerators.

**Updating Inverted Lists**
 This simple approach to inverted lists is bound to cause performance problems if the DBMS does not just sustain a read-only query load, but also has to handle updates. If some tuple is updated, deleted or inserted, costly table rearrangement of the tail-sorted BAT[oid,T] would be necessary to keep this BAT ordered on tail. Recall that BATs are dense arrays of values (without holes or slack space), so deleting some BUN in the middle would require moving all subsequent BUNS one position back, to fill up the hole. The average cost of moving half of all tuples on every update is fat too expensive. The default implementation of the MIL delete operator actually does not do this, it rather keeps the BAT dense by moving an extreme (either first or last) tuple into the deleted position, sacrificing any sorted properties the BAT holds (the exact BAT update algorithms are discussed in more detail in Section 7.4).

In order to be able to accelerate read-only queries with sorted BATs and being able to sustain updates at low cost, we use three BATs, called the S-, I- and D-BAT, to store an inverted list index on a column of a relational table:

**S-BAT[oid,T]** ordered on tail, named S.<TABLE>..<COLUMN>. It holds the column values for all tuples of the table at the last sync-point in sorted order. The tuples appear in such an oid order that the tail values form an ordered sequence. The "S" prefix stands for "sorted", as the S-BAT is a (stale) sorted copy of the C-BAT.
7.2. PHYSICAL STORAGE OF RELATIONAL TABLES

Figure 7.5: Handling Updates in H-, C-, S-, I, and D-BATs
CHAPTER 7. MONET AS RDBMS BACK-END

I-BAT[oid,T] named I.<TABLE>_<COLUMN>. It holds the freshest column value for all tuples that have been inserted since the last sync-point. The “I” prefix stands for “inserted”.

D-BAT[oid,T] named D.<TABLE>_<COLUMN>. It holds the old values for all tuples present at the last sync-point that were subsequently deleted. The “D” prefix stands for “deleted”.

This can also be seen in Figure 7.3 (the M-BATs mentioned there will be discussed later in this section).

Note that updates in the C-BAT are represented as a sequence of a delete and an insert, therefore causing an insert in both the D-BAT and the I-BAT.

The update mechanism is illustrated in Figure 7.5. Let us assume that we create an inverted list index on the TOTALPRICE column of the ORDERS table. Hence there will be three additional BAT[oid,dbl]-s:

- one called S_ORDERS_TOTALPRICE that stores all [oid,name] combinations ordered on price (notice the arrowhead drawn at the tail column in Figure 7.5),
- a I_ORDERS_TOTALPRICE holding all new value combinations for tuples that were inserted or ORDERS prices that have been changed.
- a D_ORDERS_TOTALPRICE holding all old value combinations for tuples that were deleted or ORDERS prices that have been changed.

Each time a table column is updated that has an inverted list, the inverted list has to be maintained as well:

insert for each new tuple with oid o that are inserted with column value v, the BUN [o,v] is inserted in the I-BAT.

delete the oid and the old value of the tuple is inserted into the D-BAT.

update is handled as a delete of the old value, followed by an insert of the new value.

The I- and D-BATs grow under tuple updates and inserts. In order to limit the overhead of the I-BAT and in order to effectively optimize query execution, the I- and D-BATs for all columns that have an inverted list index should fit in memory comfortably and should not become significant in size with respect to their S-BATs. Therefore, periodically (or, dynamically, when the size of the I-BAT exceeds a threshold, e.g. a 2% of the size of the S-BAT) we carry out a sync-point where all updates are carried through in the S-BAT.

Note that updates could be optimized by checking whether the current value combination [o,v_{old}] is part of the I-BAT (recently inserted), in which case it can be overwritten directly into [o,v_{new}]. This scheme would, however, involve a lookup into the I-BAT whenever an update occurs. If real-time update performance should be optimized, one could choose not to do this optimization, and process all double updates at the sync-point in bulk:

X := intersect(D_SUPPLIER_PHONE, I_SUPPLIER_PHONE);
D_SUPPLIER_PHONE.delete(X);
I_SUPPLIER_PHONE.delete(X);
7.2. PHYSICAL STORAGE OF RELATIONAL TABLES

At the sync-point, the I- and D-BATs are emptied, and a new S-BAT is created by sorting the C-BAT.

\[ \begin{align*}
S_{\text{SUPPLIER PHONE}} & := C_{\text{SUPPLIER PHONE}. \text{order}}; \\
D_{\text{SUPPLIER PHONE}} & . \text{delete}(); \\
I_{\text{SUPPLIER PHONE}} & . \text{delete}();
\end{align*} \]

Accelerating Selections with Inverted Lists

The whole purpose of maintaining an inverted list index on a relational table column is to accelerate query performance. The great asset of the inverted list is the S-BAT, which (ignoring updates) contains all column data in sorted order in a BAT[oid, any] ordered on tail, hence selection, join and group-by predicates can be executed by implementations that exploit this ordering. Coming back to a previous example, we had a selection on phone number:

\[ \begin{align*}
& \text{select NAME from SUPPLIER where PHONE = '01-425-8828080'}
\end{align*} \]

This query might hurt if the SUPPLIER table were huge and no hash-accelerator were present on the tail of C_SUPPLIER_PHONE, as the only applicable MIL implementation of the below select would be a sequential scan (see Section 5.3.4):

\[ \begin{align*}
& \text{[printf]} \text{("\%s
", select(C\_SUPPLIER\_PHONE, "01-425-8828080").access(BAT\_WRITE))}; \\
& \text{[printf]} \text{("\%s
", vl.mirror.join(C\_SUPPLIER\_NAME))};
\end{align*} \]

While a sequential scan on a single BAT is much more efficient than a sequential scan on a relational table (as only the data pertaining to one column needs to be scanned), performance can be improved using an inverted list on the PHONE column of SUPPLIER. We modify the select as follows:

\[ \begin{align*}
v{\_}l & := \text{select(S\_SUPPLIER\_PHONE, "01-425-8828080");} \\
& \text{[printf]} \text{("\%s
", vl.mirror.join(C\_SUPPLIER\_NAME))};
\end{align*} \]

This select accesses a BAT that is sorted in tail, hence can use binary search lookup. This MIL is however not complete: if the I- and D-BATs are non-empty, the S_SUPPLIER_PHONE BAT is somewhat stale. Inserted and deleted tuples are taken into account as follows:

\[ \begin{align*}
v{\_}l & := \text{select(S\_SUPPLIER\_PHONE, "01-425-8828080").access(BAT\_WRITE));} \\
& v{\_}l.\text{insert(select(I\_SUPPLIER\_PHONE, "01-425-8828080")));} \\
& v{\_}l.\text{delete(D\_SUPPLIER\_PHONE));} \\
& \text{[printf]} \text{("\%s
", vl.mirror.join(C\_SUPPLIER\_NAME))};
\end{align*} \]

We use the delete and insert instead of diff and sort as we require bag-semantics.\(^6\) As we assume that the D- and I-BATs are very small (and probably memory resident), the cost of the delete, insert and second select are all negligible.

Accelerating Joins with Inverted Lists

Inverted list indices can also be used to accelerate joins on boolean predicates like "=" , ">". Such join operators all have efficient implementations if both join columns are ordered. As the inverted list index provides just that in the S-BAT, it is apt for accelerating a join that, when executed on the standard column BAT, would have to be executed with a nested-loop implementation or would require explicit sorting first:

\(^6\)It might be more elegant to introduce specific read-only MIL bag difference and union operators in the future.
join(C_PARTSUPP_SUPPKEY, C_SUPPLIER_SUPPKEY);

If we have an inverted list for both SUPPKEY columns, the above MIL can be accelerated to:

join(S_PARTSUPP_SUPPKEY, S_SUPPLIER_SUPPKEY);

However, the above use of the S-BATs is only legal if both inverted lists are clean: their I- and D-BATs must be empty. One way to ensure this would be to force a sync-point on both inverted lists just before the join on the S-BATs. The heavy cost of executing the join on the C-BATs might justify this.

M-BATs for Inverted List Caching

In the above, we redirect MIL select calls from the C-BATs to the S-BATs because the implementation of the select uses binary search on BATs that have a sorted tail column (as described in Section 5.3.4, the default implementation uses a sequential scan), and returns a "slice-view" on the S-BAT instead of materializing a new result BAT.

While binary search is obviously more efficient than sequential scan, it still will generate roughly $\log_2(N)$ virtual memory page faults or random disk I/Os to find the desired rows (where N is the number of rows in the table). Common B-Tree implementations minimize the number of random-I/O by using very wide nodes, e.g., such that one node exactly occupies exactly one disk block. Such a tree might have a fanout in the hundreds. If we assume a fanout of 128 (e.g., with a node size of 8*128=1024 bytes), B-Tree search in a two million row table takes $\log_{128}(N)=3$ random I/Os (or page faults) to find the desired row. Simple binary search, on the other hand, takes $\log_2(2000000) = 21$ I/Os. Additionally, the top nodes of a B-tree are often kept in memory. In this example, the two top levels of the B-tree consist of 129 nodes, and 129KB can easily be cached in memory. This reduces the cost of B-Tree search to only one I/O.

Anno 2001, an overhead of twenty random I/Os can represent as much as 200 million idle CPU cycles. Therefore, we want to optimize random I/O cost to S-BATs in Monet, by applying the same technique of caching the "top of the tree" to the inverted list structures. This is done by having a fourth BAT, the so-called "memory" M-BAT, in addition to the already mentioned S-, I- and D-BATs for representing an inverted list. The tail column of the M-BAT[int,T] is constructed by taking a value from the S-BAT[oid,T] at regular intervals. Therefore, the M-BAT also has a sorted tail column. Its head column contains the row numbers where the values where taken from the S-BAT. This M-BAT is used to restrict the search-space within the S-BAT during selections, with the below scripted MIL procedure:

$$
S\_SUPPLIER\_PHONE.select("01-425-8828080");
\Rightarrow
S\_SUPPLIER\_PHONE.slice(M\_SUPPLIER\_PHONE.select(str(nil)), "01-425-8828080").max,
M\_SUPPLIER\_PHONE.select("01-425-8828080", str(nil)).min)
.select("01-425-8828080");
$$

This piece of MIL code uses the slice MIL operator to take a positional subset of a BAT. As this operator is implemented as a slice view (see Section 5.3.2) this is free and subsequent access to the result of the slice will cause access to the underlying S-BAT
that is restricted to a (small) subset. If we load(VM_SEQ) S-BAT in virtual memory, this in turn means that the I/O (page faults) will only occur for this subset.

These vanilla MIL primitives obtain the same performance behavior as a classical B-Tree with cached top levels: suppose we had constructed the M-BAT by taking a 1-in-128 sample of the S-BAT with 2 million tuples, then the M-BAT holds 16384 tuples (occupying 128KB of memory), and the selected subset will be about 64 tuples in size (occupying 64*8=512 bytes), hence the select into the restricted S-BAT just causes one virtual memory page-fault.

7.2.4 Join Index BATs

The join index [Val87] is a binary table storing “surrogate” pairs that represent a pre-computed join result. In this context, surrogates might well be represented as oid-s. A join index between tables L and R accelerates queries that require a join between L and R, by redirecting the join phase from normal join methods that use the full tables L and R to the much smaller join index representation. Join indices may be used in two directions: from L to R and from R to L. To this purpose, Valduriez advocated the use of duplicated storage of the join index in two B+ trees: one tree giving fast access on L-oid to matching oid-s from R, and another tree giving fast access in the other direction (from R-oid-s to L-oid-s).

In this example of using Monet for relational query processing, we simply consider a join index as an “artificial” data column, that directly refers to oid-s of the remote table. For example, take the foreign key relationship from PARTSUPP.SUPPKEY to the SUPPLIER.SUPPKEY column. Suppose we had an extra column called $\text{[SUPPKEY$SUPPLIER.SUPPKEY]}$7 in the PARTSUPP table that instead of carrying SUPPKEY values carries directly Monet oid-s of the matching tuples in the SUPPLIER table. Such a join-index-column saves one join step during query processing when we go from PARTSUPP tuples to SUPPLIER tuples.8

In the above example, we would get an additional C-BAT called $\text{C.PARTSUPP.=[SUPPKEY$SUPPLIER.SUPPKEY]}$. The additional column could also have been created in the SUPPLIER table as $\text{C.SUPPLIER.=[SUPPKEY$PARTSUPP.SUPPKEY]}$, but the former is more efficient in Monet as the SUPPLIER-PARTSUPP join is N-1, hence we can store the join index column in a BAT(void,oid) as a PARTSUPP C-BAT.

In order to support efficient updates and join acceleration in both directions, Valduriez proposed storage in two B+-trees. In the case of our Monet join index representation, access with PARTSUPP oid-s to the $\text{C.PARTSUPP.=[SUPPKEY$SUPPLIER.SUPPKEY]}$ is already fast, because positional lookup can be used into the BAT(void,oid). If this C-BAT is small and always fits in memory, access in the other direction can be made O(1) by using a hash table accelerator on the tail column. If this is not the case (the BAT will not be in memory always), then we can create an inverted list index on the “artificial” $\text{=[SUPPKEY$SUPPLIER.SUPPKEY]}$ column of PARTSUPP, using the exact mechanism as described in the previous section. As explained in Section 7.2.3, this leads to three additional BATs (the S-, I-, and D-BATs) named $\text{S.PARTSUPP.=[SUPPKEY$SUPPLIER.SUPPKEY]}$,

7 Just for the sake of presentation, we name join-index columns with their “mangled” boolean MIL-expression in function notation, where parentheses are substituted by brackets and commas by $\$.$

8 Notice that object oriented data models provide for such system-managed referencing between tables, by allowing explicitly specified relationships between classes (as opposed to relational systems that have to check referential integrity on table columns).
Updating Join Indices

Join indices are typically used to speed up foreign key joins. In those cases, maintenance of the join index can be integrated with referential integrity checking, which alleviates its cost. Let us elaborate with an example. If we added a new supplier, the referential integrity rules that logically follow from the TPC-H schema tell us that there cannot yet be a PARTSUPP tuple that matches this previously unseen SUPPLIER. Therefore, the join result between SUPPLIER and PARTSUPP over the SUPPKEY column will never alter under inserts of new SUPPLIER-s.

Now consider what happens when a new PARTSUPP is inserted. Part of the new PARTSUPP tuple is a SUPPKEY column, which the referential integrity checks for consistency. This check (e.g. by using a SUPPLIER.SUPPKEY.find() lookup into the inverted list accelerator) must yield a SUPPLIER oid. This oid can then be used as the value for the “artificial” = [SUPPKEY$SUPPLIER.SUPPKEY] column. Therefore, in case of a tuple insert, the referential integrity checks that are required anyway for foreign key values, already deliver the values needed for updating the join index columns.

Similarly, finding out which tuples in the foreign table are affected when an update occurs is already part of the normal referential integrity checking that an RDBMS must perform, and does therefore not count as additional join index maintenance cost. In all, join index updates follow the same (efficient) mechanism as normal C-BAT updates (see Section 7.2.2). If an inverted list is used for accelerating reverse access, the S-, I- and D-BATS for the join index column must also be maintained by the inverted list update mechanism described in Section 7.2.3.

Join Indexing N-M Relations

A special case are N-M foreign key joins. Here, tuples from each side can match zero, one or more times with tuples from the other. Therefore, we cannot use a C-BAT[void,oid] for storing the “artificial” join index column in either table. Hence, we use a C-BAT[oid,oid] to do this, where the same oid can occur 0 or more times in both columns.\(^9\) This C-BAT is kept sorted on the head column. This ensures efficient (binary search) access to oid values in the head column. An inverted list on the tail column like described above is used to accelerate join access in the other direction (to oid values in the tail column). This inverted list index is always present on N-M join indices (it is not optional).

The update protocol for N-M join indices is rather special. Since we want to keep the C-BAT sorted on head, it is treated as an S-BAT: it is not updated directly, but only brought up-to-date occasionally at inverted list sync-points. The I- and D-BATS of the artificial join column are thus used to periodically update both the S- and the C-BAT according to the inverted list update mechanism described in Section 7.2.3.

How the Join Index is Used

First, we show a simple example with a join from PARTSUPP to SUPPLIER:

\(^9\)The artificial join-index-column can be interpreted of type Set<oid>, as described in Section 4.3.1.
7.2. PHYSICAL STORAGE OF RELATIONAL TABLES

select SUPPLIER.NAME
from PARTSUPP, SUPPLIER
where PARTSUPP.SUPPKEY = SUPPLIER.SUPPKEY and
PARTSUPP.AVAILQTY > 0

The above SQL query asks for all supplier names that have a non-empty parts supply listed. The MIL query below can be used if join indices are not present:

```mil
X := select(C_PARTSUPP_AVAILQTY, 0, int(nil)).reverse;
Y := X.join(C_PARTSUPP_SUPPKEY);
Z := Y.join(C_SUPPLIER_SUPPKEY.reverse);
[printf]("XX\n", Z.join(C_SUPPLIER_NAME))
```

The first and third join are both joins into a BAT[void,T], therefore the positional-join algorithm can be used, which has minimal cost.

The second join probably is the most costly join, because the join tail column of C_SUPPLIER_SUPPKEY is not ordered (hence the MIL join implementation must first sort, or create a hash-table or a T-tree explicitly – see Section 5.3.4). However, if we have a join index on the SUPPKEY column of the PARTSUPP and SUPPLIER tables, we can eliminate this join:

```mil
X := select(C_PARTSUPP_AVAILQTY, 0, int(nil)).reverse;
Y := X.join(C_PARTSUPP_[SUPPKEY$SUPPLIER_SUPPKEY])
[printf]("XX\n", Y.join(C_SUPPLIER_NAME))
```

Let us now give an example of using the same join index the other way round:

```mil
select SUM(PARTSUP.AVAILQTY)
from PARTSUPP, SUPPLIER
where SUPPLIER.SUPPKEY = PARTSUPP.SUPPKEY and
SUPPLIER.NAME = 'Microsoft'
```

Here, the SQL query asks for the total number of parts available from Microsoft. First, we show the MIL translation without the use of join indices of this same query. This MIL is more sophisticated, as we use the inverted lists on SUPPLIER_NAME and PARTSUPP.SUPPKEY to improve performance of the select and the second join, respectively.

```mil
X := select(S_SUPPLIER_NAME, "Microsoft").reverse;
Y := X.join(C_SUPPLIER_SUPPKEY);
Z := Y.join(S_PARTSUPP_SUPPKEY.reverse);
[printf]("%%9d\n", sum(Z.join(C_PARTSUPP_AVAILQTY)))
```

Notice that the above is only correct if the inverted list on PARTSUPP_SUPPKEY is clean. However, if tuples were deleted or inserted since the last sync-point, the third statement should have been:

```mil
Z := Y.join(C_PARTSUPP_SUPPKEY.reverse).access(BAT_WRITE);
Z.insert(Y.join(I_PARTSUPP_SUPPKEY.reverse));
Z.delete(Y.join(D_PARTSUPP_SUPPKEY.reverse));
```

Again, if the join index is used, two joins can be supplanted by one:

```mil
X := select(S_SUPPLIER_NAME, "Microsoft").reverse;
Y := X.join(S_PARTSUPP_[SUPPKEY$SUPPLIER_SUPPKEY].reverse);
[printf]("%%9d\n", sum(Y.join(C_PARTSUPP_AVAILQTY)))
```

Since we use the S-BAT of the inverted list on the join index, the same argument as before can be made: if the S-BAT is not clean and updates have been recorded in the I- and D-BATs, the second statement should be corrected to:
Y := X.join(S_PARTSUPP_=[SUPPKEY$SUPPLIER_SUPPKEY].reverse).access(BAT_WHITE);
Y.insert(X.join(I_PARTSUPP_=[SUPPKEY$SUPPLIER_SUPPKEY].reverse));
Y.delete(X.join(D_PARTSUPP_=[SUPPKEY$SUPPLIER_SUPPKEY].reverse));

Given the fact that the D- and I-BATs are kept very small, the additional cost of the additional delete, insert and join operations is low.

7.2.5 Indexing Strategies For OLAP

All RDBMS implementations of the TPC-H listed in Figure 7.2, apply a similar physical design strategy, where the main tables (ORDERS and LINEITEM) have a clustered index on the SHIPDATE and ORDERDATE columns, respectively. Also, each implementation defines non-clustered indices on all (foreign) keys of each table in the TPC-H.

In the following, we discuss an indexing strategy for the TPC-H database in Monet.

Join Indices

In order to facilitate fast navigation between tables, we create join indices over all foreign key relationships, except the one between NATION and REGION (with its fixed size of 25 rows this does not make much sense). This leads to the following “artificial” oid columns:

- PARTSUPP_=[PARTKEY$PART_PARTKEY] an extra column in PARTSUPP (SF * 800.000 rows), taking 4 bytes-per tuple for the C-BAT plus 8 bytes-per-tuple for the S-BAT, hence requires SF*9.6MB.
- PARTSUPP_=[SUPPKEY$SUPPLIER_SUPPKEY], idem, hence another SF*9.6MB.
- LINEITEM_=[ORDERKEY$ORDERS_ORDERKEY] an extra column in LINEITEM (SF * 6.000.000 rows), taking 4 bytes-per tuple for the C-BAT plus 8 bytes-per-tuple for the S-BAT, hence requires SF*72MB.
- LINEITEM_and=[PARTKEY$PARTSUPP_PARTKEY]_=[SUPPKEY$PARTSUPP_SUPPKEY] another SF*72MB. This join index is built on a composed key.
- ORDERS_=[CUSTKEY$CUSTOMER_CUSTKEY] an extra column in ORDERS (SF * 1.500.000 rows), taking 4 bytes-per tuple for the C-BAT plus 8 bytes-per-tuple for the S-BAT, hence requires SF*18MB.
- CUSTOMER_=[NATIONKEY$NATION_NATIONKEY] an extra column in CUSTOMER (SF * 150.000 rows), taking 4 bytes-per tuple for the C-BAT plus 8 bytes-per-tuple for the S-BAT, hence requires SF*1.8MB.

Note that each join index is two-sided, hence consists of the “artificial column” C-BAT in the “left” table (that provides fast access to the “right” table), plus an inverted list index (S-, I, and D-BAT) for fast access from “right” to “left”. All included, the storage cost for these join indices is SF*183MB.
Clustered Indices

A clustered index actually re-orders the rows in the base table storage such that the rows appear in the order of the index column. As you can order a base table in only one way, most RDBMS software allows to define one clustered index per table. Anyway, as the TPC-H rules prohibit use of replication of table data in anything else than indices on single columns or (compound) keys, one clustered index is the most TPC-H allows.

The effect of the clustered index on selections queries is best illustrated when one considers a range-selection query on the column with the clustered index on it. This query now selects a consecutive chunk of rows from the base table. This has two benefits. First, such a selection optimizes the amount of I/O, since consecutive rows means that all rows present in the retrieved disk blocks are actually used. Second, instead of random I/O, the RDBMS can use sequential I/O, which is much more efficient. As discussed in Section 3.2.1 this performance advantage of sequential over random I/O is increasing exponentially over time.

The rationale for the RDBMS implementations to create clustered indices on the SHIPDATE or ORDERDATE columns of the LINEITEM and ORDERS tables respectively, is that most TPC-H queries have a range-constraint on these columns (or another date column that is strongly correlated to it).

Clustered indexing can also be applied in Monet. In addition to creating an inverted list on the column (e.g. LINEITEM.SHIPDATE), this is done by placing the values that are stored in all C-BATs of the LINEITEM table such that the C_LINEITEM.SHIPDATE BAT[void, date] has an ordered tail column. Range-queries on this column then have the "horizontal" advantages already enjoyed by RDBMSs (all rows in a disk block are used, and sequential I/O) plus the additional "vertical" advantage that only columns in use are retrieved. Monet truly minimizes the amount of I/O required here.

The goal of Monet is to provide ad-hoc query functionality at high performance. We admit that analyzing a fixed set of queries and concluding that two specific clustered indices enhance performance for those queries, does not fit well the ad-hoc philosophy. We believe that the vertical fragmentation and memory/CPU efficiency of Monet do enable ad-hoc querying on any predicate at high performance. However, neither do we want to put ourselves at an unfair disadvantage when comparing the benchmark results of Monet with commercial RDBMS implementations of TPC-H. Therefore, we do create clustered indices on LINEITEM.SHIPDATE and ORDERS.ORDERDATE. That is, the physical order of the rows stored in the C-BATS follows those respective columns, plus we create inverted lists (S-, I-, and D-BATs) on both columns.

Note that when handling updates to a table with a clustered index in Monet, it is not necessary to immediately reorder the entire table (all C-BATs) if an update breaks the ordering of the index column. The fact that the table is ordered only enhances the access pattern of range-selects on SHIPDATE but is not required for database consistency, and breaking the ordering in a small percentage of rows will not influence performance significantly. A full table reorder after updates have broken the ordering, could be done periodically, e.g. only at data warehouse refresh time.

The extra storage costs of these clustered indices are the two inverted lists (one of about SF*7 million values, one of about SF*1 million). The disk space needed for these BATs totals SF*68MB (each S-BAT[oid, date] is 8 bytes wide).
Non-Clustered Indices

The question now arises whether additional (non-clustered) indices should be created on additional columns, and if so, on which? Recall that the preferred indexing structure in Monet is the hash table, if the database fits in main memory. Therefore, inverted lists in Monet are relevant only when the database does not fit in main memory. This we may suppose in case of the TPC-H benchmark. We now provide a simple model, that describes the I/O cost in both Monet and a "normal RDBMS" for evaluating a generic select-project query, with or without non-clustered indices:

```
select COL₁,...,COLₚ from TABLE where COL₀ = X
```

Let \( f \) denote the fraction of tuples selected (the selectivity), and \( n \) the total number of columns of the table, and \( C \) the relation cardinality (the number of tuples). For simplicity, we assume a column-width \( w = 4 \) bytes for all columns. The query is called a select-project query as it typically consists of two phases. In the select phase, we determine which tuples are selected (this involves \( COL₀ \)), and in the the project phase, we fetch the other columns required for the query (\( COL₁,...,COLₚ \)).

We now discuss the **index-select** execution strategy that one can apply if one has a non-clustered index (e.g. an inverted list) on the selection attribute. As discussed before, the inverted list is a file of \([\text{id}, v]\) combinations that tells which tuples have the value \( v \). Because this file is ordered on the values, an equi- or range-predicate selects a consecutive range of entries from the inverted list file, where the \( \text{id} \)-s indicate which tuples are selected. We ignore for simplicity the cost of determining the start and end-point of this consecutive range (which can be optimized, e.g. with the M-BAT technique as described earlier). The I/O cost for reading the selected \( \text{id} \)-s consists of a sequential read of \( fC₂w \) bytes. This divided by the sequential bandwidth \( S \) gives the number of seconds required: \( (2fCw)/S \).

The factor 2 is caused by the fact that an inverted list stored pairs of values and object or tuple-ids (e.g. for an int column in Monet we get a \( \text{BAT}[\text{id}, \text{int}] \) of width \( 2^4 = 8 \) bytes).

After the select-phase, we need to fetch column values for the selected \( \text{id} \)-s. This can be done in two different **projection** strategies:

**random I/O** We take the list of \( \text{id} \)-s from the select phase, and use single block I/O requests (or page faults in virtual memory) to look up these tuples in the file that stores the main table ("normal" RDBMS case) or in the files that store the C-BATs of the needed columns (Monet case). It is good practice to sort the \( \text{id} \)-s first to avoid requesting a disk block twice for different tuples. The probability for a disk-block to be selected is \( 1 - (1 - f)^T \), where \( T \) stands for tuples per block. In the relational case, this is \( T_{rel} = B/(nw) \), and in the case of Monet \( T_{monet} = B/w \). The total costs of this phase for a "normal" RDBMS is the number of blocks in the relation \( Cnw/B \) times \((1 - (1 - f)^{B/(nw)})^T \) times the random access cost per block \( R \), and for Monet we get \( p(Cnw(1 - (1 - f)^{B/w}))R \).

**sequential I/O** We scan only the projected attributes, hence we get I/O costs of \( (pCw)/S \). A "normal" RDBMS must sequentially scan the entire relational table of \( Cnw \) bytes, hence its I/O cost becomes \( (Cnw)/S \).

For a "normal" RDBMS, the sequential strategy actually would not even use the non-clustered index, as evaluating the selection predicate could just as well be done during the sequential scan. In Monet, however, replacing the index scan by a sequential
scan over the C-BAT containing the selection attribute is also a possibility. This requires reading $Cw$ bytes of sequential I/O, which costs $(Cw)/S$ seconds. We call this alternative strategy which does not use any non-clustered index the full-sequential strategy.

Figure 7.6 shows the cost predictions given by the model on typical hardware of the year 2000 (up) and 1990 (down). The query modeled has one selection attribute and one projection attribute on the SF=100 LINEITEM table of TPC-H (600 million tuples). The result of the sequential scan execution strategy in a “normal” DBMS is directly given (the highest horizontal line). For obtaining the RDBMS strategy of index-select followed by projection with random access I/O, one should add the corresponding two lines (which is always dominated by random access I/O). That should also be done for both possible index-select strategies in Monet: index-select followed by sequential-scan, and index-select followed by random access. We used only one projection attribute ($p = 1$) in the query, so one can easily deduce how other values of $p$ would perform in the Monet case (simply multiply the Monet sequential
scan or Monet random I/O numbers by \( p \)). Finally, the Monet strategy of using sequential scan both for selection and projection is obtained by multiplying the Monet sequential scan line by \( p + 1 \).

These results show that for a "normal" RDBMS, it currently makes sense to use an index for accelerating selections of up to about 1 in 2000 tuples. This already very small selection percentage is rapidly decreasing with the hardware trend that I/O bandwidth follows Moore's law but I/O latency does not, as discussed in Section 3.2.1. A decade ago, when disks had a bandwidth of 1MB/s and a latency of 14ms (as opposed to 20MB/s and 7ms), the break-even point still was at about 1 in 200 (as can be seen in the right plot of Figure 7.6). Any linear rule like "use an index below 1 in 200" or "below 1 in 2000" will soon be outdated as Moore's law continues to hold (not even considering the now widespread usage of RAID devices that increase sequential disk bandwidth even further).

As a conclusion, non-clustered indices are useful for accelerating selections in a "normal" RDBMS only when a handful of tuples are retrieved. A common example of select-queries that only generate such a handful of result tuples, are equi-selects on one (foreign) key value. While that is typical for OLTP loads, in query-intensive areas like OLAP and Data Mining selection percentages are much higher, hence a sequential scan is almost always faster than a query evaluation strategy that uses an index.

In Monet, the same line of reasoning goes for using random access for retrieving projection values, the difference being that the break-even point is even earlier (about 1 in 35000, in this configuration). This is caused by the fact that a disk image of a BAT stores much more tuples per disk block than a non vertically fragmented relation, hence the probability that a disk-page is needed by the random access strategy increases.

Random I/O access in Monet, as little applicable as it is, is technically achieved by memory-mapping the C-BATs and giving the MADV\_RANDOM virtual memory advise, before accessing them with the MIL join operator. As mentioned above, this advise is only wise if very few tuples are selected. Normally, it is better to use sequential file I/O for reading the BAT, or use a memory-mapped BAT with advise MADV\_SEQ (which is default) and profit from today's high sequential I/O bandwidth.

When comparing in Monet the index-select/sequential-project strategy with the sequential-select/sequential-project strategy we remark the following:

- Since the S-BAT is roughly twice as big as the C-BAT, the amount of sequential scan volume in an index-select is twice that of a sequential-select, if all tuples are selected. Hence, index-select/sequential-project is only better when less than half of all tuples is selected.

- the amount of I/O needed in the index-select/sequential-project is \( fC2w + pCw \) bytes versus \( (p + 1)Cw \) for the sequential-select/sequential-project strategy. The I/O cost reduction when using the index-select is maximal when \( f = 0 \), and is hence limited to \( p/(p + 1) \). In other words, for an average query that involves 5 attributes (1 for select, 4 project), we can save at most 20% of I/O cost using inverted lists.

Going back to the issue of storing the TPC-H database in Monet, one "extreme" index design strategy that fits the ad-hoc philosophy would be to create an inverted list on each column (as suggested by the initial DSM proposal [CK85]). Hence S-, I-, D- and M-BATs would be created on all columns. As the I-, D-, and M-BATs are
kept very small, the extra disk storage required would come mainly from the S-BATs. Column five of Figure 7.4 shows the number of bytes per row occupied in each S-BAT. In when summing all these totals, we see that the “extreme” indexing strategy would cost only SF*1.33GB in Monet.

However, in the light of our previous analysis and the fact that the selection clauses in OLAP queries like those in TPC-H typically select much more than a handful of tuples, we conclude that all these inverted lists would only slightly enhance performance \((p/(p+1))\) while they do cause extra update and storage cost (conversely said, inverted lists on the key columns only are useful in OLTP settings). When non-clustered indices are not used (apart from those introduced by the join indices), this reduces the total disk storage for TPC-H in Monet to SF*0.92GB, which forms a sharp contrast to the SF*6GB - SF*16GB sizes observed in the commercial RDBMS implementations of TPC-H. One reason for this is that the commercial numbers include file redundancy in RAID devices, which can broadly be estimated to cause a factor 2-3 increase of data sizes. In a comparable Monet implementation, such RAID storage should also be used, but even then Monet is a factor 2-5 more efficient in storing data.

### 7.3 Query Execution

In the following we outline a simple SQL-to-MIL translator that is the heart of the SQL-speaking Monet front-end. The first step in this translation is a rather literal translation of the SQL query into a relational algebra tree, where the tree nodes are (standard) relational operators [Cod70, Dat85].

One of the assumptions here is a query plan that has already undergone strategic query optimization (i.e. it has been determined already which relational operators will be used and in which order). As this is a precondition of the translation of the relational query into MIL (which deals with physical execution), this means that strategic optimization has to do without physical details. We think this can be adequately done by taking the abstract measure tuple flow: the summation over all nodes in the query graph of the number of tuples that flow through them. In most cases, we expect the optimal physical query graph to be the one that corresponds to a logical query plan with lowest tuple flow. While obviously cases exist where a difference in the physical cost of one algorithm over the other might compensate for the difference in tuple flow, making a ‘cheaper’ physical algorithm on more tuples faster than a more ‘expensive’ algorithm on less tuples, this is not probable, and it seems safe to assume that at least the “bad” query plans are avoided this way; which has been stated as the practical goal for RDBMS query optimization [WP00]. As such, one could use any approach of the plethora of query optimization techniques available to minimize this tuple flow, e.g. using heuristic [ABC+76], random-based [GLPK95, GLPK94, PGLK97, Pel97], histogram-based [IP95, PIHS96], or even sampling-based techniques [AGPR99].

In the following discussion, we will not go into query optimization issues, neither considering the order of the relational operators (i.e. strategic query optimization, see Section 5.3.1) nor the choice of the appropriate algorithm for an operator (tactical optimization, see Section 5.3.3), but focus on how the MIL primitives are sufficient to execute SQL query plans efficiently.
CHAPTER 7. MONET AS RDBMS BACK-END

7.3.1 Relational Algebra Trees

In this section, we assume that the MIL translator works on a relational query tree and employs a fixed order of evaluation, which is *right-depth-first*\(^\text{10}\). The query tree has been produced by a query optimizer. The task at hand is to generate the corresponding MIL code that executes the query on the database tables stored in persistent C-, S-, I-, D- and M-BATs, as described in the previous Section. The query tree consists of the basic nodes `table` (for base tables), and `join`, `select`, `aggregate` and `sort` (the basic relational query processing operators), augmented with the auxiliary nodes `scope`, `var` and `relation`:

\[
\begin{align*}
\text{RELATION} & = \text{relation}(\text{id}ent, \text{scope}([\text{var}(\text{id}ent, \text{EXPR}<\text{any}>)], \text{TABLE})) \\
\text{BASE} & = \text{relation}(\text{id}ent, \text{scope}([\text{var}(\text{id}ent, \text{EXPR}<\text{any}>)], \text{table}(\text{id}ent))) \\
\text{OPERATOR} & = \text{join}(\text{EXPR}<\text{bit}>, \text{BASE}, \text{RELATION}) \\
& \quad | \text{select}(\text{EXPR}<\text{bit}>, \text{RELATION}) \\
& \quad | \text{aggregate}(\text{EXPR}<\text{any}>), \text{RELATION}) \\
& \quad | \text{sort}(\text{EXPR}<\text{any}>, \text{RELATION}) \\
\text{EXPR}<\text{T}> & = \text{id}ent.\text{id}ent \mid \text{const}<\text{T}> \mid f(\text{EXPR}<\text{any}>, \ldots, \text{EXPR}<\text{any}>):\text{T}
\end{align*}
\]

The above is BCNF extended with two notations:

- \([X]\), which means a list of zero or more elements \(X\).
- typed clauses \(Y<Z>\), meaning \(Y\) of type \(Z\).

The latter construct is used to denote expressions of a certain MIL type. As embodied by the last rule, expressions are either column references or literal MIL constants (see Section 4.1), or simple MIL-operators; i.e. those \(f(\ldots \text{any} \ldots):\text{any}\) that have a simple (non-BAT) return type and parameters. Column reference expressions consist of a relation identifier, a dot and a column identifier. Base relations (those `relation` nodes with a `table` grandson, as produced by the `BASE` rule) have as initial allowed set of column identifiers all column names of the persistent table.

Additional column names can be added by the `scope` nodes, which introduce a new column defined by an expression using the `var` clause. Column references in expressions in the query tree may reference columns from the current `relation` as well as from any `relation` that is reachable downward in the tree, without going past any `aggregate` nodes.

Note that columns introduced by `scope` nodes ease the elimination of common subexpressions. This task, as we indicated before, belongs to the query optimizer.

The basic relational operator nodes have bread-and-butter query processing semantics:

- `select(\text{boolExpr}, \text{rel})` return a subset of all tuples in \(\text{rel}\) for which evaluating \(\text{Expr}\) gives true.
- `join(\text{boolExpr}, \text{leftRel}, \text{rightRel})` returns a new tuple for each combination of tuples from \(\text{leftRel}\) and \(\text{rightRel}\) for which evaluating \(\text{boolExpr}\) yields true.

\(^{10}\)We use linear, non-bushy, join trees to indicate a clear sequential join order. The right-deep order is chosen for notational purposes only.
aggregate\( (exprList, rel) \) returns one tuple for each unique combination of expression results after evaluating the expressions in \( exprList \) for all tuples in \( rel \).

sort\( (exprList, rel) \) returns all tuples of \( rel \), but in lexicographical order of expression results after evaluating the expressions in \( exprList \) for all tuples in \( rel \).

The root of the tree is a relation, whose table represents the result of the query. This table is made up by all tuples in the relation, where the tuples consist only of those columns mentioned in its scope list.

The join1 node is a special case of the join node, which has an additional semantical constraint that the join between the right and left relation (its third and second parameters) couples exactly one right tuple to each left tuple. Certain join conditions with this property can be determined by analyzing foreign key relationships stored in the data dictionary. Each basic node in these relational algebra trees forms a "virtual" relational table with its own set of tuples; however join1 nodes do not add or remove tuples to/from the relational table of its rightmost parameter, they just may add columns to the table. In the case of MIL query plans, this (rather common) condition allows to save a number of processing steps; and that is why we distinguish between join1 and join nodes.

As an example, we use query 9 of TPC-H, which is among the most costly queries in many RDBMS implementations. Its SQL syntax is as follows:

```sql
SELECT NATION.NAME AS NATION,
       EXTRACT(YEAR FROM ORDER.ORDERDATE) AS YEAR,
       LINEITEM.EXTENDEDPRICE * (1 - LINEITEM.DISCOUNT) - PARTSUPP.SUPPLYCOST * LINEITEM.QUANTITY AS AMOUNT
FROM PART, SUPPLIER, LINEITEM, PARTSUPP, ORDERS, NATION
WHERE LINEITEM.SUPPKEY = PARTSUPP.SUPPKEY AND LINEITEM.PARTKEY = PARTSUPP.PARTKEY
  AND LINEITEM.ORDERKEY = ORDER.ORDERKEY
  AND LINEITEM.SUPPKEY = SUPPLIER.SUPPKEY
  AND SUPPLIER.NATIONKEY = NATION.NATIONKEY
  AND LINEITEM.PARTKEY = PART.PARTKEY
  AND PART.NAME LIKE '%green%'
GROUP BY NATION, YEAR
ORDER BY NATION, YEAR
```

Figure 7.7 shows a translation of TPC-H query 9 into relational algebra using a right-deep join plan. In the graphical representation of the same plan depicted by Figure 7.8, the relation nodes are painted as grey boxes in the background, with the relational operator nodes they directly encapsulate drawn over them. The tree is annotated with new column identifiers introduced by scope nodes at the positions where these occur.

MIL is generated from a relational algebra tree by traversing the basic relational operators in a right-depth-first order and generating MIL for each node. In the following, we will discuss exactly how this code generation works for each kind of node (including the use of join indices – which are mentioned explicitly in Figure 7.8).

### 7.3.2 Select

The select node identifies the tuples that satisfy a boolean expression tree. Let us assume in the following that the tree has a depth of more than one, i.e. that at least
the root node is a MIL operator (as the select has a boolean expression, it must be a MIL operator that returns the type bit).

This basic expression evaluation algorithm executes the expression tree left-bottom up, generating for each intermediate node (which is a MIL-operator expression f(…)) the multi-join map version of that operator (f(…)).11 The leaf nodes are either constants, which are generated as-is as parameters to the multi-join maps, or column references. For each column, we have a BAT[void, any] available. In the case that the select works on a base relation, these are the C-BATs. In other cases, these BATs are constructed using the basic projection algorithm, which is discussed later on. Each multi-join map yields again a BAT[void, any] result, which serves as parameters to multi-join maps higher up the tree.

The BAT[void, bit] that results from running the basic expression evaluation algorithm on the root expression, is then turned into an oid-list by performing a select(true) on it. This produces a subset BAT we call the selection-BAT, that has all oid-s of the selected tuples in its head column. The selection-BAT is then fed into a mark.reverse in order to produce the pivot BAT[void, oid]. The pivot is a central concept in MIL query processing; its head column contains one oid (in densely ascending order) for each tuple of the new relation, where each tail contains the oid of the original relation that produced it.

Suppose we have a base table TAB with column COL and the selection expression 42.0 < foo(TAB.COL,1.0) and TAB.COL ≥ (4*10+2), where foo() is some arbitrary simple MIL operator. The generated MIL by the basic select code generation is:

\[ v1 := [\text{foo}(\text{C\_TAB\_COL},1.0); \]

Figure 7.7: Relational Algebra Notation of TPC-H Query 9

---

11 If all parameters to the operator are constants, the expression counts as a constant which can be evaluated by simply executing the MIL operator - without multi-join map. In this way, constants are eliminated bottom-up.
An improvement over this approach is to use the built-in select range- and equiselect operator, as it directly generates a BAT[oid, any] result that can serve as so-called selection-BAT. This code is generated for the root node or for interior nodes when all operators higher in the expression trees are either or, and or not, and the operator is either =, >, <, ≤, ≥, or is a range condition, that is and(>, <), and(≥, <), and(>, ≤) or and(≥, ≤). In case of an interior node, the upper and/or/not nodes generate intersect(mirror(left), mirror(right)), union(mirror(left), mirror(right)) and diff(mirror(param), mirror(col))

Taking this modified strategy into account, we would generate the following MIL:

```plaintext
v2 := [≤](42, v1);
v3 := *(4,10);
v4 := +(v3, 2);
v5 := [≥](C_TAB_C0L, v4);
v6 := [and](v2, v5);
v7 := v6.select(true);
v8 := v7.mark.reverse;
```

The second parameter to `diff` is a column from the underlying relation that can be chosen at random. For caching purposes, it is best to re-use the last used table column for this purpose.
One detail that needs to be mentioned for this strategy is expression normalizing such that simple comparisons have the column reference as left parameter and the constant as right parameter (this involves turning \((42, X)\) into \(\geq(X, 42)\), etc.). This order of parameters is required when using the MIL select operator.

As a final refinement to the select node code generation, the SQL front-end should make use of the inverted list indices when present to enhance performance of range selection expressions on base columns. If we assume that TAB.COL has an inverted list and thus a sorted S-BAT and an in memory M-BAT summary plus I- and D-BATS with recent notifications, the following MIL should be generated:

As an optimization in those select nodes that emit just one MIL select from a C- or S-BAT to arrive at the selection column, we have the selected column values already in the tail the select-result. If this column is being referenced in an expression upward in the algebra tree, the basic projection algorithm (described below) does not need to join into the C-BAT holding it, but use the tail column of the select-result instead, with its head column newly mark-ed.

For completeness, we discuss what happens if the select expression does not have a MIL-operator as root of the expression tree. One possible case is that the expression is a simple column reference of type boolean. In that case, we can skip the basic expression evaluation phase, and directly select the true values to arrive at the selection BAT. The other case is when the selection expression is a constant expression (i.e. one without column references at all) of type bit. In that case, we generate the following code for generating the pivot in v02:

\[
\begin{align*}
v1 := & \text{...code for computing constant omitted...} \\
\text{if} \ (\text{not}(v1) \text{ or isnil}(v1)) \{ \\
\quad v2 := \text{bat}(\text{void}, \text{oid}); \ # \text{empty BAT} \\
\text{else} \{ \\
\quad v2 := \text{C\_TAB\_COL.mirror}; \\
\}
\end{align*}
\]

### 7.3.3 Basic Projection Algorithm

When expressions are evaluated, column references must be resolved into column-BATs.

For each vertex in the tree between two relation nodes, the code generation for the father node computes a so-called pivot BAT[valoid], which relates oid-s in the father relation (head column of pivot) to oid-s in the child relation (tail column of pivot). This was already described in the case of the select node, and will also be described for the other kinds of nodes (note that the binary join node produces two pivots, one for each join relation).
The basic projection algorithm just joins all pivots on the path between the relation node where the expression occurs (source) and the destination relation node of the column reference, starting at the source downward. As these actions are all of the form $\text{join}(\text{BAT}[\text{void,oid}], \text{BAT}[\text{void,oid}])$, the efficient Positional-Join algorithm gets used. Even better, by virtue of the radix-accelerator described in Section 6.5 such joins are transparently accelerated when necessary by the cluster-decluster cache-optimized join strategy.

The end result of these joins is a combined, “path”, pivot $\text{BAT}[\text{void,oid}]$ that relates oid-s from the expression-node to the destination node, deeper in the tree. If the column definition is a column of a base table, we can directly join the “path” pivot with its C-BAT in order to obtain the desired projection column. Note also that if the source relation is the direct father of the base relation, we do not need to create the “path” pivot as we can use right away the “single-vertex” pivot created by the father.

If the destination, however, is a new column introduced by a var expression in the scope clause of the destination relation, this same projection algorithm is recursively used to create a projection BAT for that scope column (if the code for that column had not been generated before).

One can see that this basic algorithm can be enhanced by pivot re-use. The most obvious form is to re-use “path” pivots for all projection columns over the same path. A second form is re-using earlier created “path” pivots for creating pivots over a superset of that path. For example, if node C refers through node B to a column in base relation A, it will construct a “path” pivot C-A by joining C-B and B-A. Then, if relation D refers through C and B also to A, the pivot D-A is constructed by joining D-C with the earlier created C-A. As we consider right-deep trees only, a simple check for already generated “path” pivots along the path between column reference and definition (and choosing the longest one if there are multiple pivots eligible) provides a good degree of re-use already.

7.3.4 Join

The most common join expression is simple equi-join between two single columns. This is directly supported by the MIL join operator. Suppose we have tables RIGHT and LEFT that both have a column KEY and we have a join expression $=(\text{RIGHT.KEY,LEFT.KEY})$. This generates the following MIL:

```mil
# the basic equi-join
v0 := join(C_LEFT_KEY, C_RIGHT_KEY.reverse);

# dynamically optimized partial cluster
v0 := v0.sql_joincluster(C_LEFT_KEY.count, <ln>, <lw>, C_RIGHT_KEY.count, <rn>, <rw>);

# pivot creation
v1 := v0.mark.reverse;
v2 := v0.reverse.mark.reverse;
```

Here $v_1$ is the pivot between the join node and the right node, and $v_2$ the pivot between the join node and the left node.

In Section 6.5.2 we showed that if the number of tuples in one of the join relation is so high that its C-BAT’s exceed the memory cache size, Positional-Joins with random access generated by the basic projection algorithm, will thrash the memory cache. A solution to this problem for one of the two relations is to partially cluster the join result on the oid-s of that relation. This should be done before creating the pivots.
This tactical optimization can be performed by the below \texttt{sql\_clusterjoin} MIL procedure, that dynamically figures out whether the subsequent computation of column projections can be accelerated by partially clustering the join result. The variables \texttt{<ln>}, \texttt{<lw>}, \texttt{<rn>} and \texttt{<rw>} follow from the query plan and can be emitted as constants directly by the SQL front-end.

\begin{verbatim}
PROC sql\_joincluster(
    BAT[oid,oid] joinresult,
    int left\_count, left\_nproj, left\_maxwidth
    int right\_count, right\_nproj, right\_maxwidth) : BAT[oid,oid]
{
    IF ((left\_nproj > 0 and left\_count*left\_maxwidth > CACHE\_SIZE/2)
        or (right\_nproj > 0 and right\_count*right\_maxwidth > CACHE\_SIZE/2))
    {
        IF (left\_count*left\_nproj*left\_maxwidth > right\_count*right\_nproj*right\_maxwidth) {
            VAR nbits := 1 + \log2((left\_count*left\_maxwidth)/CACHE\_SIZE);
            VAR nignore := min(0, (1 + \log2(left\_count)) - nbits);
            VAR npasses := 1 + (nbits-1) / CACHE\_LINES;
            RETURN joinresult.radix.cluster(npasses, nbits, nignore);
        }
        ELSE {
            VAR nbits := 1 + \log2((right\_count*right\_maxwidth)/CACHE\_SIZE);
            VAR nignore := min(0, (1 + \log2(right\_count)) - nbits);
            VAR npasses := 1 + (nbits-1) / CACHE\_LINES;
            RETURN joinresult.reverse.radix_cluster(npasses, nbits, nignore).reverse;
        }
    }
    RETURN joinresult; # no clustering necessary
}
\end{verbatim}

The above MIL procedure takes into account the number of \textit{direct} column references that the join node makes to its join parameter relations, \textit{as well as} the sizes of these two relations, and the maximum BUN-width of the C-BATs involved (this can be determined by looking at the column types). Partial clustering is only done if at least one of the relations has a projection column that does not fit half of the cache. Which relation to cluster on is then heuristically selected by comparing the "projection volumes" (the product of size, width and number of projection columns) of both relations. Calling of this \texttt{sql\_clusterjoin} is done as a post-processing step after any "real" join code discussed in this section, except the exactly-one \texttt{join1} case.

Notice that applying the cluster-decluster join strategy on the projections of the other relation is is done automatically when needed by virtue of the \textit{radix-accelerator} and therefore does not need explicit code generation.

\section*{Multi-Column Equi-Join}

The advantage of MIL is that its operators can be implemented for great efficiency, as they have a operator signature with a low degree of freedom. The downside is that this low degree of freedom is achieved by processing data column-at-a-time, which complicates multi-column operations. In the case of multi-column equi-join -- this happens e.g. when keys consist of multiple columns -- one should combine all columns involved on both sides of the join into numerical (preferably \texttt{int}, but also \texttt{long}) columns, and then equi-join those two numerical columns. The best way to do this is to use a \textit{perfect} mapping function \texttt{f(...any...):int} that maps each combination of values on a predictable number, and numbers are never assigned to multiple combinations.

In practice, such mapping functions for numerical types yield a numerical type that has a bit-width which is larger than or equal to the sum of the bit-widths of all types of the columns involved, and we use simple bit-shifts and -ors to combine values efficiently.
in a non-overlapping way. This mapping strategy can be best executed by a relational algebra tree rewrite. The script below gives an example of a two-column (both int) equi-join that is transformed into an equi-join on single-column lng-s:

```
join(and(=LEFT.KEY1,RIGHT.KEY1), =LEFT.KEY2,RIGHT.KEY2),
  relation(LEFT, scope(S1,BASE))
  relation(RIGHT, scope(S2,REL)))
⇒
join(=LEFT.NUM,RIGHT.NUM),
  relation(LEFT, scope(S1+[NUM=xor(hash(LEFT.KEY1), hash(LEFT.KEY2))],BASE))
  relation(RIGHT, scope(S2+[NUM=xor(hash(RIGHT.KEY1), hash(RIGHT.KEY2)],REL)))))
```

When a perfect mapping function does not exist, one can use the MIL operator hash(any):int on all non-integer column-BATs using the multi-join map [hash](t>), and [xor](a,b) all the resulting integers together. This mapping function may be a good hash function, but not a perfect hash function, therefore in the resulting join of integer columns, *false hits* may occur. These then have to be filtered out. Again, we give an example of this strategy as a relational algebra tree rewrite of a two-column equi-join into a single-column equi-join on non-perfectly mapped hash integers, followed by a select to filter out false hits:

```
join(and(=LEFT.KEY1,RIGHT.KEY1), =LEFT.KEY2,RIGHT.KEY2),
  relation(LEFT, scope(S1,BASE))
  relation(RIGHT, scope(S1,REL)))
⇒
select(and(=LEFT.KEY1,RIGHT.KEY2), =LEFT.KEY2,RIGHT.KEY2),
  relation(TMP, scope[]),
  join(=LEFT.NUM,RIGHT.NUM),
  relation(LEFT, scope(S1+[NUM=xor(hash(LEFT.KEY1), hash(LEFT.KEY2)],BASE)),
  relation(RIGHT, scope(S2+[NUM=xor(hash(RIGHT.KEY1), hash(RIGHT.KEY2)],REL)))))
```

A special property of the MIL hash() function is that it never sets the highest bit, hence the xor() cannot set it either, hence the resulting combined NUM values can never by accident form the special int(nil) value (which would lead to missed join hits due to the nil-semantics).

**Generic Join Expressions**

The generic fall-back code generation for join expressions that are not equi-joins is to loop over the right relation and for each tuple use the select node code generation to select tuples from the left relation. This is done by generating a scripted MIL procedure on-the-fly that receives as parameters the current right oid plus all referenced columns from right relation (and its sons — to obtain column BATs for those the basic projection algorithm is used first). Inside the body of the MIL procedure goes the select code that generates the selection BAT. In this code generation, all references from columns in the right relations (or its sons) are remapped to the MIL procedure parameters.

```ml
v1 := bat(oid,oid);
proc tmpproc(oid o, any c1, ..., any cn) : bat[oid,oid] {
  var selbat := ... use select-node code generation for selecting in left relation ...
  v1.insert(selbat.project(o).reverse);
}
[tpmproc](right_coll.mirror, right_coll, ..., right_col1);
undef tmpproc;
```

```ml
v2 := v1.mark.reverse;
v3 := v1.reverse.mark.reverse;
```
A possible refinement is to first pre-process the right relation with an aggregate in order to obtain all unique elements. This reduces the size of the outer relation in the above nested-loop join. The normal aggregate algorithm would then be extended with code generation for an aggregation pivot. As a post-processing step, the join result would then be joined with the pivot in order to bring back the previously eliminated doubles.

**Join Indices**

One thing to check, however, before any join code is generated, is whether a join index exists for the join expression. This should be looked up by comparing the join expression to those in the data dictionary.\(^{13}\)

In the simple case of a join between two base relations, the join-index BAT can be used right away as join result $\text{BAT}[\text{oid}, \text{oid}]$. In our right-deep trees, the left relation is always a base table, but the right table may be an inner node. If that is the case, one first needs to create a pivot to the base table that forms the other end of the join index. This base table can be reached through some path in the inner node (this can be done as described in the basic projection algorithm). The join result in those cases is then formed by joining the pivot with the join index.

Notice that depending on how the join index is defined in the data dictionary, one needs to join against the C-BAT or its reverse. In the latter case, the join may be more efficient if one uses the inverted list $\text{S-BAT}$ instead, which has an ordered tail column.

Figure 7.8 shows that almost all joins in TPC-H Query 9 are accelerated by join indices. Those join nodes have a gray border, and their expressions are depicted in gray as to indicate that these expressions and joins need not be executed.

**Exactly-One Joins**

In the case where we join a right relation with a foreign key to a left relation with an unique key on the equality of those keys, we can already deduce from the information in the data dictionary that for each right tuple, we will find exactly one left tuple.

Such joins can in the Monet context be seen as simple projections, as they just serve to add columns to the right relation (they do not remove or introduce new tuples). This in turn means that renumbering and pivot joining is not necessary; therefore $\text{join}$ nodes are treated differently than normal $\text{join}$ nodes.

After computing the join result $\text{BAT}[\text{oid}, \text{oid}]$ with the normal methods, it is sorted with $\text{reverse.order.reverse}$ to make sure the head $\text{oid}$-s appear in densely ascending order.\(^{14}\) This is just done to enhance performance of later processing of this BAT. This join result is not marked, but can be used as a pivot for projecting in columns from the left table using the basic projection algorithm.

\(^{13}\)Some normalization of join expressions is required to recognize all matching join index expressions easily.

\(^{14}\)We know that the $\text{oid}$ sequence must be densely ascending when sorted, because all $\text{oid}$-s of the right relation form a $\text{void}$ column, and we perform joins that hit exactly once only, hence we end up with the same $\text{oid}$ collection.
Outer Joins

Any join or join node can be marked as an “outer-join” in a certain direction, meaning that if a tuple from the source side does not match any from the destination, one extra result is emitted, that has all nil values in its projections columns from the destination.

In the case of equi-join, outer-join is supported directly by using the MIL outerjoin instead of join.

Join indices are just as applicable (as they are only a mechanism to re-use join results). The generic join code can support outer-join with just a small addition after the selbat computation:

```java
if (selbat.count = 0) selbat := bat(oid,oid).insert(o,nil);
```

The basic projection algorithm needs to be slightly adapted to use outerjoin instead of join for those pivots known to be outer-joins (this is a transitive property for “path” pivots: if it consists of at least one outer-join pivot, itself is one as well).

Finally, in the case of multi-column equi-join with a non-perfect hash function, one needs to modify the “false-hits” filter expression:

```java
and(-(LEFT.KEY1,RIGHT.KEY1) , (LEFT.KEY2,RIGHT.KEY2))
```

As the NUM field is computed with xor() of hash() which never yields a nil, we know that any nil values that do occur are introduced by outer-join projections, and thus should not be filtered out (hence the added or() condition).

7.3.5 Sort

A sort node only makes sense as root of the algebra tree, because it just changes the output order when visualizing the query result. It appears in a relational algebra tree when a SQL query has a ORDER BY clause, but also when it has a GROUP BY clause but defines any aggregate expressions. In those cases, any additional ORDER BY columns that are not in the GROUP BY, are appended to the column list of the sort node.

In order to sort a relation on one column BAT[void,any], a tail-sorted version of this is produced by order. The resulting BAT[oid,any] with oid-s appearing out of order in the head we call the global ordering BAT. This global ordering BAT is mark.reversed to produce the desired pivot BAT[void,oid] which relates tuples in the new order to tuples in the old order.

The case of multi-column orderings is an extension of the computation of single-column orderings. After creating the single-column global ordering BAT on the first ORDER BY column with the unary order, the remaining ORDER BY columns are processed iteratively with binary order operations, to refine this global ordering, with the previous global ordering BAT as first and the ORDER BY column-BAT as second parameter.

As described in Section 5.3.5, the binary order is in fact just a MIL-procedure, that marks the global ordering BAT on both sides, creating PIVOT and GROUPING BATs, that have a new void head and contain respectively the tuples oid-s and the order values in their tail. The additional ORDER BY column is then provided with these new dense head oid-s, by joining it with the pivot. As such, the COrder can walk sequentially
through all its three BAT parameters. Per existing group, it Quick-Sorts the oid-s and new ORDER BY column values in order to emit the refined global ordering BAT.

These details of the tertiary \texttt{CTorder} (see Section 5.3.5) are depicted in Figure 7.9, also including the cluster-decluster implementation of the join between PIVOT and the ORDER BY column (used automatically thanks to the \texttt{radix-accelerator}, see Section 6.5.2). This implementation of multi-column ORDER BY thus is decomposed in basic MIL operators that all have cache-friendly access pattern: the possibly costly reordering of each ORDER BY is eased by cluster-decluster, whereas the refinement sorting uses the cache-friendly Quick-Sort on sub-chunks of the global ordering BAT (which are much smaller than the total relation and often directly fit the memory cache).

One potential problem in MIL is that most of its operators are set- or bag-defined and thus guarantee nothing about ordering. This is also the case with the multi-join map, of which we use \texttt{[printf]} to print result tables. In principle, the issue of presenting results in the correct order is considered a front-end issue. Therefore, for queries with ORDER BY we add an additional first index column to print by \texttt{[printf]}, which contains the row number. The row number is a unique integer with minimal value 0 – for the tuple that has to come first – and maximal value $N - 1$ – for the tuple that has to come last – where $N$ is the number of tuples in the result). In practice,
the printed result is not necessary, as the MIL implementation of the multi-join map where all BAT-parameters have a void column, will output tuples in the order of the densely ascending oid-s.

### 7.3.6 Aggregate

Relational aggregate nodes are introduced whenever the SQL query contains a GROUP BY clause with aggregate functions in the project list, or if the SELECT DISTINCT keyword is used. In the latter case, all SQL projection columns appear both as GROUP BY columns and in the projections (in that case there are no "real" aggregates - the algorithm is the same, though).

Aggregates are computed by first creating a so-called grouping-BAT\([\text{void,oid}]\) where the tail values divide all oid-s into a number of disjunct groups. With that grouping-BAT, the MIL pump construct \(\{f\}(\text{EXTENT}, \text{GROUPING}, \text{COLUMN})\) can compute aggregate expressions. COLUMN is a column-BAT\([\text{void,any}]\) holding a column or expression on the child relation, and EXTENT is a BAT\([\text{void,any}:1]\) with the new table oid-s in the head and in the tail the unique collection of group values (from the tail of GROUPING BAT\([\text{void,oid}]\)).

We mentioned that an aggregate algebra node cannot be passed by projections to fetch column values deeper in the tree (therefore, it is the only algebra node that does not need to construct a pivot). All available columns are the scope expressions which must have an aggregate functor in their root. The only non-aggregate expressions allowed in this scope clause are the GROUP BY expressions themselves (the first parameter of the aggregate node).

There are two algorithms to compute the pivot from a set of GROUP BY expressions, available initially as column BAT\([\text{void,any}]\) and are produced by the basic projection algorithm. The first pivot algorithm uses a successive execution of the MIL group operators on the GROUP BY column BATs, first the unary group on the first BAT, then binary group-s with the result of the previous group as first parameter and the next column BAT as second parameter. The final result of this process is the desired BAT\([\text{void,oid}]\). Notice that the group has the special property that the group-oid-s that it puts in the tail column, are taken from the collection of head values from the group members. Choosing the oid-s in this way, rather than e.g. assigning new unrelated numbers, makes it easy to project the GROUP BY expressions into the result of the aggregate node: they are simply computed by joining the extent BAT\([\text{void,oid}]\) with the GROUP BY expression column BAT\([\text{void,any}]\).

The second pivot algorithm uses the code generation for sort algebra nodes to compute a global ordering BAT\([\text{oid,oid}]\) from the GROUP BY expressions as if they were ORDER BY expression. Using mark-s, on both sides the earlier described PIVOT and GROUPING are computed. The latter BAT\([\text{void,any}]\) is used as grouping-BAT when aggregates are computed with the MIL pump (the second parameter). The column-BATs passed as third parameter to the pump are formed by joining the PIVOT with the projected column-BATs. Finally, as the tail-values of GROUPING are newly numbered, they cannot be used directly for projecting through the GROUP BY expressions. We therefore compute an additional map-BAT\([\text{void,oid}]\) with \(\{\text{min}\}(\text{EXTENT}, \text{GROUPING}, \text{PIVOT})\) that can be used for joining against the column-BATs holding the GROUP BY expressions.

The first pivot creation algorithm uses the MIL group operator which is implemented internally using simple and fast hash-grouping. The second algorithm uses the order
operator, which is based on repeated chunk-wise Quick-Sort. As a general rule, the first algorithm will be much faster as long as the number of groups is limited (i.e. as long as the hash table fits the memory cache). The second algorithm has a memory access pattern that is cache-friendly, even with a huge amount of groups\(^\text{15}\) and therefore outperforms the first in such cases.

A typical case where many different groups are expected is when the aggregate node was generated for a SELECT DISTINCT SQL query. In other cases, statistics kept in the data dictionary, such as histograms of attribute distributions could be used to estimate the number of different groups for a GROUP BY expression in order to choose between the two pivot generation algorithms. Dynamic query optimization is an attractive alternative, though, as it comes as no additional cost in MIL and does not depend on modeling or estimation, but on the actual value distributions. Here, one would pass an extra integer parameter to the MIL group-s, that holds maximum memory consumption for the hash-table used while grouping. A good value for this would be like half the size of the most costly cache, as determined by the Calibrator (see Sections 6.3.3). If this version of the group encounters too many different groups, it just quits and returns bat(nil)\(^\text{16}\). So, after the group-s, the generated code would have the form:

\[
grouping := \text{group}(\text{coll}, \text{CACHE}\_\text{SIZE}/2).\text{access(BAT}\_\text{WRITE});
\]

\[
gru\text{oup} := \text{group}=(\text{grouping}, \text{col1}, \text{CACHE}\_\text{SIZE}/2);
\]

.. if (isnil(grouping)) {
  ..code generated by sort-based aggregate pivoting ..
} else {
  ..rest of code generated by group-based aggregate pivoting ..
}

.. processing continues with the projections of the aggregation node ..

Here we use the :f=(x,..) MIL assignment notation, which is a shorthand for x := f(x,..), because there is a specific implementation of :group=(b) that places its results directly in the input-BAT b, hence optimizes (cache) memory usage.

Note that if the (sort-based) pivot generation is used and the relation node above the aggregate node is a sort and its ORDER BY columns are a subset of the GROUP BY columns, then the aggregate result is already properly sorted, so that no code needs to be generated anymore for the subsequent sort.

### 7.3.7 Generating MIL For TPC-H Query 9

We now do a right-depth-first traversal of the query tree in Figures 7.8 and 7.7, showing which code is generated. In the rightmost operator of the algebra tree there is a simple select that is executed with a multi-join map [like]. Thus, for relation R6, we compute as pivot v02.

\[
v00 := \text{[like]}(\text{C\_PART\_COLOR}, \":\text{green}\")
\]

\[
v01 := v00.\text{select(true)};
\]

\[
v02 := v01.\text{mark.reverse};
\]

---

\(^\text{15}\)This cache-friendliness includes the use of the radix-cluster/decluster join strategy for joining column BAT's with the partial ordering, as discussed at the end of the last Subsection.

\(^\text{16}\)The binary group(b,v) also returns bat(nil) if isnil(b)
The join between R1=LINEITEM and R0=PART on their PARTKEY columns is accelerated by a join index (simply denote here "t1-t0"). As the join in R6 is not a base table join (the PART appears below a select), we must join the pivot of relation between relations R7-R0 (v02) with the join index R0-R1. As we come from the side of PART and the join index is stored in LINEITEM, we use the inverted list version of the join index for faster access. Therefore, code for taking into account updates in D- and I-BATs is included.

Given the fact that the rest of the query only projects columns from the LINEITEM side, we can follow a (c,u) projection strategy (as mentioned in Section 6.5.3) thus clustering the join result on some H most significant bits.

The join result R6-R1 stored in v03 represents the relation R7. Using mark-s on v03 we obtain the pivot between relations R7-R6 in v04 and the pivot between relations R7-R1 in v05.

\[
v03 \leftarrow v02 \Join (S_{LINEITEM}=[\text{PARTKEY}_\text{PART}] \text{.reverse}) \text{access} (\text{BAT_WRITE});
\]
\[
v02 \text{.insert}(v02 \Join (I_{LINEITEM}=[\text{PARTKEY}_\text{PART}] \text{.reverse}));
\]
\[
v02 \text{.delete}(v02 \Join (D_{LINEITEM}=[\text{PARTKEY}_\text{PART}] \text{.reverse}));
\]
\[
v03 \leftarrow v03 \text{.sql_joincluster}(v02 \text{.count}, 0, 0, S_{LINEITEM}=[\text{PARTKEY}_\text{PART}] \text{.count}, 2, 4);
\]
\[
v04 \leftarrow v03 \text{.reverse.mark.reverse};
\]
\[
v05 \leftarrow v03 \text{.mark.reverse}.
\]

The sql_joincluster in this case partially clusters the join result on LINEITEM, as there are only projections from that side. Notice that R7 in fact references three columns from R1=LINEITEM (to be exact T1.SUPPKEY, T1.PARTKEY, and T1.ORDERKEY), but these expressions do not count as they appear in join-expressions that are handled with join indices. However, each use of a join index counts as one oid column projection, hence we have two column projections of C-BATs with byte-width 4.

What follows now are a four joins with ORDER, PARTSUPP, SUPPLIER, and NATION of which all but the latter can all be handled with join indices. As these are all exactly-one joins, everything happens in the context of column projections inside relation R7.

Column projections are performed by lazy evaluation, which starts when a column is referenced in an expression. This first happens when evaluating the scope list of the result of the ORDER join, where the year is computed from the R2.ORDERDATE. Thus, we take the pivot R7-R1 and join it with the join-index between R1=LINEITEM and R2=ORDER. The order mentioned in the join1 code generation algorithm is free here, as we join into a BAT[void,oid] join-index (it is a C-BAT), hence the result is another BAT[void,oid] (as mentioned before, a void column in the left operand is propagated by the Positional-Join). The basic projection algorithm then continues to produce a column with date tails in v07, and the basic expression algorithm computes a column-BAT[void,int] with extracted years.

\[
v06 \leftarrow v05 \Join (C_{LINEITEM}=[\text{ORDERKEY}_\text{ORDERKEY}]) \text{.order};
\]
\[
v07 \leftarrow v06 \Join (C_{ORDER.ORDERDATE});
\]
\[
v08 \leftarrow \text{[year]}(v07);
\]

We then proceed to execute the three remaining joins of relation R7 in order of appearance. The joins with R3=PARTSUPP and R4=SUPPLIER are accelerated by join indices LINEITEM-PARTSUPP ("R1-R3") and PARTSUPP-SUPPLIER ("R3-R4"). In order to project the NATIONKEY column from R4 a pivot R7-R4 is constructed by taking R7-t1 and joining through the join indices R1-R3 and R3-R4 in v10. The join with NATION is executed without join index in v12. As this also is an exactly-once join, re-pivoting is not necessary, and all this still happens in the context of projecting columns for relation R7.
CHAPTER 7. MONET AS RDBMS BACK-END

What follows are projections for expressions from the aggregate expressions of relation R8. Here we use the multi-join map-overwrite [:f=] whenever a map result has the same signature as its first parameter and this first parameter is not used further on. The advantage of this overwrite is that less memory is consumed and hot cache lines are better re-used. The disadvantages is that the generated code gets bit cluttered with access calls in order to make the BATs updatable.

As we used enumeration-types in some C-BATs storing the LINEITEM table, the projection joins return a BAT[void,enum1[dbl]] in v14 and v18. The single-BAT multi-join maps in v15 and v19 then fall into the enumeration-view implementation, which is virtually free, because it only processes the enumeration mapping-BAT (see Section 5.3.2).

The evaluation of R8 continues with generating the "GROUPING"-BAT in v24 with two successive group-s. The unique group oid-s are collected in v25, and mark-ed in v26 to form the extent. The GROUP BY columns (year and name) are projected using this extent in v28 and v27. This amount aggregate is finally computed with a MIL pump in v29.

The amount of tuples in R8 is small, hence the execution cost of the rest of the MIL code is insignificant. What we have here is a two-column ORDER BY. Notice that the binary order in v31 is a MIL procedure that uses the tertiary order implementation that includes the custer-decluster strategy, as depicted in Figure 7.9. The result is the final global ordering BAT, which is mark-ed to create the pivot in v32.

The statements v33–v35 are Positional-Joins that project the final three columns, which are subsequently printed using a multijoin map.

We now also show the code generated for the alternative aggregate pivot creation algorithm, which is sort-based. In the case of the TPC-H data distribution, this
algorithm will be more costly than the one showed before, as the number of groups in for a GROUP BY NATION, COLUMN is limited (tens of tuples).

Statements v23–v26 are mostly identical to v30–v32 from the other query plan. In v27 the “GROUPING”-BAT is computed, in v29 the extent and in v30 the map-BAT.

```plaintext
v23 := order(v13);
v24 := order(v23, v08);
v26 := v24.mark.reverse;
v27 := v24.reverse.mark.reverse;
v28 := v24.reverse.mirror.unique;
v29 := v26.mark.reverse;
v30 := {min}(v29, v27, v26);
v31 := v30.join(v13);
v32 := v30.join(v08);
v33 := {sum}(v32, v27, v26);
{printf}("%9d %9d %5.2f\n", v30.mirror, v31, v32, v33);
```

As the ORDER BY columns of the subsequent ORDER node are equal to the GROUP BY columns of the aggregate node, the result of relation R8 in BATs v31–33 is already properly ordered such that no specific ORDER code needs to be emitted for R9.

### 7.4 Transaction Management

The simplest transaction system based on MIL would use a global read-counter protected by a short-term lock, and a write-lock. The read-counter serves to count the number of executing read-transactions, where the first concurrent reader acquires the write-lock, and the last to finish releases it. Write-transactions acquire this write-lock, modify the C-, I- and D-BATs according to the scheme in Figure 7.5 and use the global commit to atomically commit all changed BATs before releasing the write-lock again.

This simple scheme, however, is inefficient because:

- write-transactions are costly, as saving each modified BAT at least costs two I/Os (one BAT descriptor, one heap).
- there is no concurrency at all between write-transactions, even if they obviously read and modify independent data.
- write-transactions seriously impair the performance of read-queries due to the coarse locking protocol.

In this Section, we outline a more efficient transaction system with ACID properties [GR91] implemented purely in MIL, that uses a two-level locking protocol, exploits virtual memory techniques for Isolation at low resource cost, and achieves Atomicity and Durability with write ahead logging [Dav73] and uses BATs to implement differential files [SL76]. Its aim is to provide highly concurrent write-transactions in combination with high performance read-only query execution on a nearly up-to-date database version. Figure 7.10 shows the extensions to the data dictionary from Figure 7.3 that are made in order to facilitate this transaction system. In the following, we describe how it provides ACID properties. First, we cover the features needed for consistency and isolation, then those needed for atomicity and durability.
class DatabaseVersion {
    relation Set<ReadTable> readTables inverse ReadTable::version;
    relation Set<WriteTable> writeTables inverse WriteTable::version;
    lock commitLock;
    integer transId;
}

class ReadTable:Table {
    relation DatabaseVersion version inverse DatabaseVersion::readTables;
}

class WriteTable:Table {
    relation DatabaseVersion version inverse DatabaseVersion::writeTables;
    lock readLock, writeLock;
    integer nReaders;
    BAT r_bat; // [oid,trans] combinations for read-locked tuples
    BAT w_bat; // [oid,trans] combinations for write-locked tuples
}

class ReadColumn:Column {
    lock refCntLock;
    integer refCnt;
}

class WriteColumn:Column {
    lock columnLock;
    BAT i_bat; // deltas since last sync (inserts)
    BAT d_bat; // deltas since last sync (deletes)
}

class Transaction {
    relation Set<Transaction> waitsFor inverse Transaction::waitsOn;
    relation Set<Transaction> waitsOn inverse Transaction::waitsFor;
    lock barrier;
    Set<WriteColumn> modifications; // private not yet committed column deltas
    BAT h_bat; // [table,oid] deleted tuples in this transaction.
    BAT n_bat; // [table,oid] inserted tuples in this transaction
}

Figure 7.10: Data Dictionary Extensions for Transaction Management

7.4.1 Consistency and Isolation

Consistency is achieved by a simple two-level locking protocol. Isolation is provided by making sure that a transaction does not make changes to those parts of BATs that may be accessed by other concurrent transactions.

Supporting Multiple Database Versions

Instead of the database being the collection of all tables, another level of abstraction is added by the “database version”. The database hence stores a collection of all database versions. Such a database version in principle contains a copy of all tables, columns, keys, indices, etc. The principle motivation behind this is that there is one database version – the one with the highest transaction ID – that is the most fresh “write-version” of the database. All modification queries must run against this database version and follow the transaction mechanism in order to acquire ACID properties. Then, there is one “read-version” that is kept up-to-date with the write-version at regular (small) time intervals. Read-only queries thus have the possibility to bypass all transaction overhead and run with full concurrency (i.e. without any locking) against this slightly older version of the database.

The fact that multiple versions of the same database exist, means that each database
version references its own C-, S-, I-, D-, M- and H-BATs in the Monet back-end. The inverted list structures can in fact be shared between read- and write-version, so only the H- and C-BATs are duplicated between the two database versions. To distinguish between those, we write \( H_r \)-BAT and \( C_r \)-BAT for those referenced by the read-version and \( H_w \)-BAT and \( C_w \)-BAT for those referenced by the write-version. Most of the duplication is between the \( C_r \) and \( C_w \)-BATs as these are voluminous, but by exploiting OS virtual memory primitives (\texttt{mmap} with the \texttt{MAP_PRIVATE} flag) the memory and disk consumption of this duplication is reduced to only those virtual memory pages where the \( C_w \)-BATs actually differ from their \( C_r \)-BAT counterparts.

Once in a while, a new read-version is created by propagating the current write-version to it. This happens in the background on a private copy, so read-queries need never be interrupted. When the old read-version is replaced by the new read-version, the old read-version is scheduled for deletion by setting its transaction ID to \texttt{nil}. Its BAT-resources are deleted as soon as the last still running read-queries that use them have finished.

In the data dictionary, the write-version of the database only contains references to “write-tables”, whereas the database read-version(s) only refer to “read-tables”. A write-table is different from a read-table in that it contains structures for the locking protocol. Similarly, “read-columns” are only referred to by read-tables and “write-columns” are only referred to by write-tables. Write-columns reference an \( I_w \) and \( D_w \)-BAT that holds updates to the column that have happened in the write database version since the read-version. In the case of columns with an inverted list, the \( I_w \) and \( D_w \)-BATs are supersets of the I- and D-BATs of the inverted list (we use the same virtual memory page sharing technique as in the \( C_w \) and \( C_r \)-BATs to minimize resource consumption).

The full BAT data structures of the transaction system are shown in Figure 7.11.

A Simple Locking Protocol

Read-columns add a reference-count and a lock to protect it, to support the garbage collection mechanism for old read-versions. The idea here is that each read-only query increases the reference count of the database columns it is going to use, before starting query execution. When a read-only query finishes using a table column for the last time, it decreases the reference count of its column locks. When a column reference count reaches zero and the database version is to be deleted (it has a \texttt{nil} transaction ID), the column is deleted. If it was the last column, the old database version itself is deleted.

Explicit locks on the MIL level are introduced by a simple extension module:

```plaintext
MODULE lock;
ATOM lock;
OPERATOR create_lock() : lock;
OPERATOR destroy_lock(lock)
OPERATOR set_lock(lock)
OPERATOR unset_lock(lock)
OPERATOR try_lock(lock) : bit
  ...semaphores etc...
END lock;
```
The idea here is that the data dictionary and other data structures are stored in BATs and MIL variables in the running Monet server, the entire transaction system is programmed out in MIL. We describe here a simple two-level locking protocol: a table-level read-write lock (where multiple readers or a single writer are allowed). For a certain class of queries, i.e., those that have (at least) an equi-select condition on one table key, or (at least) a foreign-key equi-join condition on each base relation\(^\text{17}\), we lock on the level of tuples (allowing multiple tuple readers or a single writer). Multiple such tuple-level locked transactions can run at the same time. A transaction can run as long as its locks do not conflict with those of already running transactions. Tuple-level read-locks conflict with the table-level write-lock, and tuple-level write-locks conflict with both read- and write-lock on the table level.

To implement this scheme, write-tables have a read/write lock (implemented in Figure 7.10 as a nReaders counter protected by a read-lock, and an exclusive write-lock that is set by update transactions or by the first reader (and released by the last reader). For

\(^{17}\text{In such queries, it is easy to establish which tuples participate, hence classical problems like "ghost records" and complex solutions like "predicate-locks" do not play a role.}\)
tuple-level locking, two BATs called the R-BAT and W-BAT store oid-s of tuples that are read-locked and write-locked, respectively. Concurrent updates to these BATs are protected by the (short-term) read-lock. Also, there is a waitsFor/waitsOn dependency graph between transactions for implementing locking and deadlock detection.

Notice that a transaction consists of one or more queries. Each query can be either a read-only (SELECT...FROM...WHERE) or a modification query (i.e. DELETE, UPDATE, or INSERT). The level of transaction locking (tuple or table) is re-evaluated for each newly arriving query in the transaction, but can only escalate from tuple- into table-locking (not the other way around). Locks acquired by a query in a transaction are only released after the transaction has finished (i.e. committed or aborted).

A running transaction directly updates the H-, C-, I- and D-BATs as depicted in Figure 7.5. To be precise, it updates the Ht-, Cw-, It- and Dw-BATs. The latter two are private delta BATs found in a write-column object. Such objects are created on-the-fly for a transaction as it modifies a column. Consequently, when transaction queries use an inverted list and it has modified that column already, it must use the union(If миллион, Iw миллион) and union(Dt миллион, Dw миллион) where otherwise the I- and D-BAT had been used. The Ht-BAT is a private version of the H-BAT that holds the oid-s of those tuples that have been deleted by the transaction. In fact, all tuple deletes from all tables are recorded in the same Ht-BAT, the tail holding the table name. Similarly, each transaction maintains a Nt-BAT that holds all oid-s of new tuples (the tuples the transaction intends to insert).

For handling a query, that is part of some transaction protected with tuple-level locking, we run the query first in “optimistic” mode without carrying out any updates, just collecting from it the “read-set” and “write-set” of tuples (i.e. those tuples from base tables that have been read, and those that are about to be deleted or modified). This is called the “read-part” of the query. The BATs holding the resulting read- and write-sets are intersect-ed with the R- and W-BATs of their respective write-tables (thus MIL also proves itself useful for implementing the locking protocol). If there are transactions (the “dependents”) that already hold conflicting locks, a vertex from the executing transaction to each of those other transactions is created. After creating the vertices, a cycle-detection on the transaction graph is run to detect deadlocks, in which case the new transaction is aborted. A transaction with dependents blocks on its lock, only to be unlocked by the last dependent that finishes executing and removes its locks from the R- and W-BATs and removes its incoming dependency vertices. Once unlocked, the blocked transaction tries again to run the query in optimistic mode, which repeats until this yields no conflicts. If no locking-conflicts arise, the read-set and write-set of the transaction are added to the R- and W-BATs and the second “write-part” of the query is executed which uses MIL update statements to make modifications in the write-version of the database.

In the following, we describe in more detail how this works for the various kinds of modification queries.

DELETE

Starts with a relational algebra tree that represents a query with a result that exactly forms a unique key on the table where deletes must be done. The semantics of such a delete query is that the tuples identified by the retrieved key values are deleted. We add a join to the base table between the base table and this query result, that retrieves
the values of all columns of the tuples that are to be deleted:

\[
\text{relation}(\text{DELETE}, \text{scope}(\{X.\text{KEY}\}, \ldots \text{XYZ}\ldots))
\]

\[
\Rightarrow
\]

\[
\text{relation}(\text{DELETE}, \\
\text{scope}(\{\text{base\_table.COLL}, \ldots \text{base\_table.COLn}\}, \\
\text{join}(=(\text{base\_table.KEY}, X.\text{KEY}), \\
\text{relation}(\text{base\_table, scope}([], \text{table}(\text{TABLE}))), \\
\text{scope}(X.\text{KEY}), \ldots \text{XYZ})))
\]

After executing this query in MIL, there will be a PIVOT between DELETE and base\_table. Also, for all columns in the scope list, there will be a column-BAT[void,any] holding all column values that are about to be deleted. These query results are used as follows for performing the deletes:

\[
\text{deletes} := \text{PIVOT.reverse}; \\
\text{Hw\_TABLE.insert(deletes.mark(nil))};
\]

\[
# \text{for each column-BAT[void,any]} \text{COLi} (1 \leq i \leq n) \text{ in DELETE} \\
\text{Dt\_TABLE\_COLi.insert(deletes.join(C\_DELETE\_COLi))} \\
\text{Cw\_TABLE\_COLi.replace(deletes.project(nil));} # \text{is omitted in leave as-is-policy}
\]

This adds the oid-s of all deleted tuples to the Hw-BAT, and for each column inserts all deleted column values to the Dt\_BATs. For columns with nil-replacement policy, all tail values of deleted tuples are replaced by \text{nil} in bulk.

**INSERT**

We assume an arbitrarily complex relational algebra tree that defines a result table. Its tuples need to be inserted in a base table, where columns are simply matched on name (omitting columns to be filled with NULL values).

\[
\text{relation}(\text{INSERT}, \text{scope}(\{Y.\text{COLl}, \ldots Z.\text{COLn}\}, \ldots \text{YZ}\ldots))
\]

This yields column-BAT[void,any]s for each column in the table (for columns omitted in the scope of the INSERT we use COL1.project(nil)).

\[
\text{set\_lock(LOCK\_TABLE)}; \\
\text{inserts} := C\_INSERT\_COLi.mark; \\
\text{fills} := \text{inserts.join(Hw\_TABLE\_COLUMN.mark.reverse)}; \\
\text{appends} := \text{inserts.mirror.diff(fills.mirror).mark(Cw\_TABLE\_COLi.count)}; \\
# \text{remove holes that are filled up now} \\
\text{Hw\_TABLE\_COLUMN.delete(fills.reverse.mark(nil))}; \\
\text{unset\_lock(LOCK\_TABLE)};
\]

\[
# \text{add newly inserted tuples to the Nt-BAT} \\
\text{Nt\_TABLE\_COLi.insert(col\_fills.reverse.project("TABLE"))}; \\
\text{Nt\_TABLE\_COLi.insert(col\_appends.reverse.project("TABLE"))}; \\
# \text{for each column-BAT[void,any]} \text{COLi} (1 \leq i \leq n) \text{ in INSERT} \\
\text{col\_fills} := \text{fills.reverse.join(C\_INSERT\_COLi)}; \\
\text{col\_appends} := \text{appends.reverse.join(C\_INSERT\_COLi)};
\]

\[
\text{set\_lock(LOCK\_TABLE\_COL)}; \\
\text{Cw\_TABLE\_COLi.replace(col\_fills)}; \\
\text{Cw\_TABLE\_COLi.insert(col\_appends)}; \\
\text{unset\_lock(LOCK\_TABLE\_COL)};
\]

\[
\text{It\_TABLE\_COLi.insert(col\_fills)}; \\
\text{It\_TABLE\_COLi.insert(col\_appends)};
\]
In the above code, we first couple oid-s of the ins.query result table with new oids in the target base table. Two BAT[oid,oid]-s are constructed, one called fills, filling in holes in the base table, and the result of the inserts ends up in append, appended at the end of the C-BATs of the base table. The holes are then deleted from the H omega-BAT of the base table, and for each table column the inserts that go into holes are processed with bulk-replace-s in C-BATs, and the append with bulk-insert-s. Both changes are also appended to the I r-BATs.

Notice that inserts use the MIL set_lock() and unset_lock() primitives to manipulate the locks referred to by the data dictionary: the short-term table lock is required when determining the oid-s for the new tuples. For each write-column, the column-lock must be held for serializing the column-BAT inserts (we assume that each transaction inserts into the table columns in the same order).

**UPDATE**

Like in the case of delete, we assume an arbitrarily complex relational algebra tree that defines a result table that forms a unique key the base table, but also contains the new values of all columns that are updated. This relational algebra query is rewritten into a join back to the

\[
\text{relation}(\text{UPDATE}, \text{scope}([X.\text{KEY}, Y.\text{EXPR1}, \ldots, Z.\text{EXPRn}], XYZ...))
\]

\[
\Rightarrow
\text{relation}(\text{UPDATE},
\text{scope}([\text{base_table.COL1}, \ldots, \text{base_table.COLn}, X.\text{EXPR1}, \ldots, Y.\text{EXPRn}],
\text{join1}=(\text{base_table.KEY}, X.\text{KEY}),
\text{relation}(\text{base_table}, \text{scope}([], \text{table(TABLE)))),
\text{scope}([X.\text{KEY}, Y.\text{EXPR1}, \ldots, Z.\text{EXPRn}], XYZ...)))
\]

This creates a PIVOT to the base table, as well as column-BATs that hold for all columns the old values as well as the new ones. This translates in the following MIL:

```
# for each column-BAT[void,any] COLi (1 <= i <= n) in UPDATE
Dt_TABLE.COLI.insert(PIVOT.reverse.join(C_UPDATE.COLI));
newvals := PIVOT.reverse.join(C_UPDATE_EXPRi);
It_TABLE.COLI.insert(newvals);
Cw_TABLE.COLi.replace(newvals)
```

For all modification queries, the read-set of a modification is formed by taking the tail column of all pivots to the base tables. In addition to that, for all columns that form a foreign key and that are used in the query, those oid-s reachable by joining the pivot with the foreign-key column are part of the read-set of the tables to which the foreign keys refer. The write-sets are the head columns of the MIL variables deletes, fills and newvals from the descriptions above.

### 7.4.2 Atomicity and Durability

Atomicity is achieved by a commit mechanism that either writes all changes made by a transaction to a persistent medium (in case of commit) or none (abort). Durability is provided by a crash-recovery mechanism that ensures that all changes, and only those changes, made by committed transactions are present when the database gets back on-line. A central component in achieving both is a Write Ahead Logging (WAL) mechanism implemented in MIL on top of the file-I/O extension module, of which we show the interface:
CHAPTER 7. MONET AS RDBMS BACK-END

MODULE io;

ATOM fp;

OPERATOR fopen(str filename, str mode) : fp // open a file for read/write
OPERATOR fclose(fp) // close an open file
OPERATOR ftrunc(fp) // truncates an open file at the current position
OPERATOR ftell(fp) : int // returns the current file position
OPERATOR fseek(fp, int pos) : bit // moves the current file position (may fail)
OPERATOR fflush(fp) : bit // flush writes; returns success status
OPERATOR fprintf(fp, str, ..any..) : bit

..other standard I/O commands ..

// MIL marshalling
OPERATOR getValue(fp) : any // read a MIL value from the stream
OPERATOR putValue(fp, any) // write a MIL value to the stream
END io;

The module provides basic file I/O with limited MIL value marshalling functionality, as the getValue() and putValue() manipulate token streams of MIL values. Such marshalling can be done by first outputting the MIL atom type as an 32-bits integer, the byte-length of the value as another 32-bits integer, followed by the bytes that make up the value. As such, the MIL-value can easily be reconstructed when reading bytes from the stream. Simple values as int can be marshalled in this way, but also BAT[any, any]-s.\textsuperscript{18}

The simple WAL structure writes into one sequential file, that contains a marshalled sequence of values that adhere to the following regular expressions:

\[
\text{WAL} = \text{ TRANSACTION* } \\
\text{ TRANSACTION} = \text{ H<BAT> N<BAT> TABLE_DELTA* transactionID<int> } \\
\text{ TABLE_DELTA} = \text{ tablename<str> COLUMN_DELTA* } \\
\text{ COLUMN_DELTA} = \text{ columnname<str> I<BAT> D<BAT> }
\]

This states that the WAL is a sequence of transaction-deltas, that each consist of a H- and N-BAT, followed by a number of table-deltas and a transaction-ID. A table-delta consists of a table-name followed by a number of column-deltas. A column-delta in its turn consists of a column name and an I- and D-BAT. Transactions in the WAL have densely ascending transaction-IDs. The transaction-ID serves as a validator in case the database might crash while writing the WAL. The presence of a correct transaction-ID determines whether the last transaction in the WAL counts as committed or not.

**Commit**

When a transaction tries to commit, it must carry out the following steps:

1. grab the global database write-version lock.

2. write all its changes to the WAL. This determines whether the commit succeeds. A failed WAL-write must truncate the WAL to its original position before entering the abort sequence. Any of this or the following steps failing should stop the database and enter the crash recovery algorithm.

3. increment the transaction ID in the database write version.

\textsuperscript{18}Marshalling is a bit more complex as that the head and tail types must also be marshalled, and the binary image of the BATs consists of all used areas of all its heaps plus the attached column properties. Recursive BATs are not supported in this marshalling.
4. release the global database write-version lock.

5. add all column-deltas in the I- and Dt-BATs to the Iw- and Dw-BATs of the database write-version. These MIL-insert-s are done while holding the column-lock of the respective write-columns.

6. add all oid-s of tuples it deletes to the respective Hw-BATs. These MIL Hw_TABLE.insert(Ht.select(tablename)) operations must be protected by holding the short-term table-lock).

7. grab again the global database write-version lock.

8. remove all incoming waitson vertices in the transaction dependency graph, and wake up any transactions for which that was the last dependency.

9. release the global database write-version lock.

Notice that the Hw-, Cw-, Iw- and Dw-BATs are inserted into by committing transactions, while other queries for transactions that are still ongoing read their stable elements without locking.\textsuperscript{19}

**Abort**

In case of an abort, all modifications of the transaction are undone by:

```c
# for each table with inserts or deletes
deletes := Ht.select("TABLE").mark;
inserts := Ht.select("TABLE").mark;
set_lock(LOCK_TABLE);
Hw_TABLE.insert(inserts);
Hw_TABLE.delete(deletes);
unset_lock(LOCK_TABLE);
# for each modified TAB.COL
set_lock(LOCK_TABLE_COL);
Cw_TAB_COL.replace(Dt_TAB_COL);
unset_lock(LOCK_TABLE_COL);
```

The abort then continues with step 7 as in the commit sequence.

**Stabilizing a New Read-Version**

When creating a new stable version, new BAT disk images for all modified columns need to be made. This is also the time when inverted list indices are reorganized once their I- and D-BATs get too big. Consequently, this process can take some time, so it would be highly undesirable to take either the read- or the write-version off-line while this process lasts.

\textsuperscript{19}In order to make this fully safe, we ensure that no BAT-reallocation in the Hw-, Cw-, Iw- and Dw- takes place by reserving space for a large number of tuples when they are created (which is supported efficiently on the OS level by anonymous virtual memory with reserved pages that are committed on demand), and by triggering any overflow that does occur by having all these BATs in access(BAT_UPDATE) mode, and testing the result status of all insert-s and replace-s. In these fail (due to reallocation needs), the transaction lock level is escalated to prevent reads concurrent with a reallocation, and the transaction commit is resumed at the insert or delete that failed. The MIL procedure that does this is omitted for brevity in the examples.
Taking the read-version off-line is prevented by creating full new disk images independently (in different disk files) from the current read-version. This is supported directly in Monet using its management of persistent BATs with global commit. Thanks to the read-column garbage collection mechanism, the switchover can be made instantaneously, without need to wait until all read-queries on the old read-version have finished.

As for the write-version, off-line time is minimized in the following scheme that consists of three phases:

**Initialization** First, we grab the global write-version lock and a table-level write-lock on all modified tables. Then, we record all sizes of the \( I_w \)- and \( D_w \)-BATs and make a copy of the \( H_w \)-BATs into what we will call "\( H_n \)-BATs". We then read the current transaction-ID, and release the global lock and all table-level write locks again.

**Reorganization** One by one, we process all modified columns. This starts by making a modifiable copy of the \( C_r \)-BAT with \( \text{copy}(\text{Cr}_\text{COL}.\text{access}(\text{BAT}.\text{WRITE})) \) into what we call a "\( C_n \)-BAT". As the \( C_r \)-BAT is read-only and loaded virtual memory (i.e. \( \text{load}(\text{VM}.\text{NORMAL}).\text{access}(\text{BAT}.\text{READ})) \), Monet will use privately mapped into virtual memory, meaning that the OS will never write modifications to the disk images, rather create private temporary page copies of the modified or appended pages in the swap file.

We then apply all changes in the transaction subranges of the \( I_w \)- and \( D_w \)-BATs to this \( C_n \)-BAT by using \( \text{Iw}_\text{TAB}_\text{COL}.\text{slice}(10, \text{hi}) \) and \( \text{Dw}_\text{TAB}_\text{COL}.\text{slice}(10, \text{hi}) \). The \( \text{hi} \) value is the size recorded in the initialization phase. The \( \text{lo} \) and \( \text{count}(\text{I-BAT}) \) resp. \( \text{count}(\text{D-BAT}) \) for columns with inverted list (0 otherwise), so we get only those column-deltas that were added since the last read-version was stabilized. We then make the \( C_n \)-BAT persistent under a new name (by e.g. prefixing all BAT names with the current transaction-ID). If the column has an inverted list, we create an "\( I_n \)-BAT" and "\( D_n \)-BAT" as \( \text{slice}(0, \text{hi}) \) of the \( I_w \)- and \( D_w \)-BATs and make them persistent under a new name. However, if these \( I_w \)- and \( D_w \)-BATs are too large, a new \( S_n \)-BAT is recreated by sorting the \( C_n \)-BAT and making it persistent under a new name. In that case, empty \( I_n \)- and \( D_n \)-BATs are created. After processing all columns this way, and giving all new BATs mode \( \text{access}(\text{BAT}.\text{READ}) \), a consistent new read-version has been stabilized, and it is mapped into virtual memory with \( \text{load}(\text{VM}.\text{NORMAL}) \). Lastly, the transaction-ID of the new database version is set, and the transaction-ID of the old read-version is changed into nil. To make these changes and all new BAT-images permanent, the BATs holding the data-dictionary are committed using the global commit. From that moment on, new read-only queries use the new read-version, whereas the old read-version will be garbage collected as soon as the last read-query executed on it finishes.

**Synchronization** We get back at the write-version and grabbing the table-level write locks and the global lock again. Further updates are halted after acquiring these locks. In the meantime, possibly some transactions may already have committed since the initialization phase. Thus we change all \( I_w \)- and \( D_w \)-BATs \( X \) into \( X.\text{slice}(\text{hi}, X.\text{count}) \), where \( \text{hi} \) was the BAT-size recorded during the initialization phase. Also, all \( C_w \)-BATs are destroyed, and recreated from the
new $C_w$-BATs with $\text{copy}(\text{Cr.TAB.COL}).\text{access}(\text{BAT.WRITE})$, again using privately mapped virtual memory. These new $C_w$-BATs are then brought back up-to-date by reapplying the just modified $I_w$- and $D_w$-BATs to them. After processing all columns in this way, the global lock and all table-level locks are released.

Notice that modification queries are only halted during the short initialization and synchronization phases, while they can continue undisturbed during the lengthy reorganization phase.

**Crash Recovery**

When the database starts up, the database write-version can be reconstructed by starting out loading all $H_w$-, $C_w$-, $I_w$- and $D_w$-BATs from their $H_r$-, $C_r$-, $I_r$- and $D_r$-BAT disk images with $\text{copy}(\text{Cr.TAB.COL}).\text{access}(\text{BAT.WRITE})$ to obtain independent privately mapped copies. For columns that do not have an inverted list, the $I_w$- and $D_w$-BATs are created empty.

Then, the WAL is scanned for the range of transactions with IDs larger than the transaction ID of the stable database version. For each such transaction, the table- and column-deltas are read, de-marshalling the $H_t$-, $N_t$-, $I_t$- and $D_t$-BATs from the bytes in the file, which are then used to re-do all modifications using the normal transaction commit protocol (except, of course, the writing to the WAL). After processing all committed transactions in this way, the write-version of the database is brought online again.
7.5 Conclusion

In this chapter, we have shown that it is possible to build a full-fledged RDBMS on top of Monet as a SQL-2 front-end that uses MIL for all its data management needs, including transaction processing. We have also outlined how the implementation in Monet achieves efficient performance for such a system.

Concerning data storage, we described how relational tables can be stored in BATs, in such a way that efficient read- and write-access is possible. In this discussion, we also included the use of two index structures: the inverted list and the join index. Here, we provided a small I/O cost model, showing that for all queries but those that return a handful of tuples (i.e. OLTP queries), the exponential advantage that I/O bandwidth gains over latency causes any non-clustered index structure to be useless regardless of its characteristics – the problem being the projection phase that follows the selection phase in query processing. In contrast, the vertical fragmentation employed in Monet attacks the real problem as it reduces projection costs, by allowing a query to only access those columns that are actually used and compressing these columns with enumeration types.

The practical case used as an example throughout this chapter is the TPC-H benchmark, which nicely fits the Monet focus on query-intensive areas. As for overall storage costs, the disk volume occupied by Monet including indices, tends to roughly equal the nominal database size, which is considerable lower (at least a factor 2-5) than any known RDBMS implementations of TPC-H.

As for query processing, we described full relational query translation into MIL, rather than just some canonical examples. This was formalized by defining standard but powerful relational query algebra trees, and providing algorithms for generating (efficient) MIL from them.

Finally, we described a transaction processing scheme built purely with MIL that provides master-slave copies of the data, where the master is the "up-to-date" database image, and the slave is a slightly old version that can be used for complex read-only queries. As Monet will never be an OLTP champion due to its vertical fragmentation, such a mixed usage scenario is relevant as it is conceivable that one uses Monet for strong OLAP performance, but also needs to accommodate updates. While all techniques employed (differential files, master-slave copies, multi-level locking with deadlock detection and write-ahead logging) have been known for decades, the novelty of this scheme lies in the Monet implementation that provides this functionality by relying on shared virtual memory rather than a buffer manager, and cleanly separates transaction functionality from its query algebra, while in other DBMSs these are heavily intertwined. The advantage of the Monet approach is that its query algebra primitives can be optimized more efficiently and analyzed more cleanly (e.g. on their memory cache behavior). The drawback that this places some burden on the MIL user to invoke explicit locking primitives is insignificant as MIL is not intended as an end-user language, but is typically generated by front-end systems.