Business System 12

Notes keyed to slides of the presentation given at
TTM Implementers' Workshop
University of Northumbria, 2-3 June 2011

Hugh Darwen

No one who cannot rejoice in the discovery of his own mistake deserves to be called a scholar.
— Don Foster

1. Cover slide

"Where R means R" is perhaps a little bold. BS12 was "truly relational" in the light of the general understanding of Codd's model at the time but is far from conforming to *The Third Manifesto.*

The Don Foster quote is cited by Ron Rosenbaum in *The Shakespeare Wars* (Random House, New York, 2006). Foster had made a name for himself by claiming to have proved that a certain dreadful poem from the early 17th century was by William Shakespeare. His "proof" was based on an analysis of word frequencies in Shakespeare's known opus—*using a database!*—and was at first accepted by many Shakespeare scholars. Foster wrote that sentence in an article in which he cheerfully accepted the refutation that eventually came.

In this presentation, which is inevitably somewhat autobiographical, I try to rejoice in the *re*discovery of some of my own worst mistakes; I also rediscover some good ideas (none of which were my own) for possible consideration by designers of *TTM*-conforming database languages. I also perhaps over-indulge in a personal trip down memory lane, in which case I just hope you'll understand and forgive any perceived excesses.

2. A Brief History of Data

This slide and the next are taken from my university lecture titled *The Askew Wall.* The notes for those slides are copied here.

In the days of cards and tapes, all files had to be accessed sequentially, starting with the first record and then repeatedly moving on to the next one until the last one had been processed. This involved the coding of *loops* in application programs, *nested* loops when more than one file was needed , as was usually the case. The algorithms involved were quite tricky and a common source of *bugs* in the application programs.

The advent of disks allowed us to dispense with some of the loops but introduced the equally trap-laden process of *pointer chasing.* Disks also raised the possibility of making an organisation's data a *shared resource* for concurrent access by a possibly diverse community.

Codd recognized both the opportunity and the dangers. He sought a foundation for technology that would allow application programs working with what were later to be called *databases* to avoid both the use of loops and the use of pointers.

The line for 1975 mentions two languages, ISBL and SQL. ISBL was a brilliant piece of work done by a small research team in the United Kingdom. A notable omission from Codd's papers was any precise definition of a computer language that would materialize his idea. The ISBL team made good this omission, faithfully adhering to Codd's model. The System R team working in the USA had a slightly different objective: to show that Codd's idea could be implemented efficiently enough for commercial use even in the "on-line transaction processing" (OLTP) environment. But the System R team of skilled "engine-room" workers didn't seem to understand Codd's idea as well as the ISBL team and arguably weren't very good language designers either.

It isn't always the good work that achieves high visibility and acclaim.

3. A Brief History of Me

Terminal Business System (TBS) was an early DBMS, before that term had even been coined. Working on it, as a fairly junior programmer to begin with, I learned about the problems of database management and the typical solutions that were known about in those days. But Terminal Business System applications had to use loops and chase pointers. We included a couple of simple scripting languages for maintaining the database and generating reports, but in each case a script could access no more than one file and required the file to have records of uniform format. These scripting languages were called, prosaically, File Maintenance and Report Writer. Users complained about their limitations. File Maintenance scripts couldn't include checks for consistency with other files in the database, or even among different records in the same file; and Report Writer scripts couldn't combine information from several files.

So that's why my attendance on Chris Date's course in 1972 was a "personal watershed". His account of the new idea from E.F. Codd gave us the ideal solution to the very problems our customers were begging us to solve. And I had been working, in vain, seeking possible solutions that very year. The real solution that now presented itself as far as Report Writer was concerned was beautifully simple: just replace the input file by a relational query result and leave everything else the same.

Six years later it was time to develop a new DBMS and by now the relational model had attracted enough attention to make it the obvious choice. We looked around the research world to see what it had made of Codd's idea and discovered ISBL, which not only solved all of the problems in language design that we had been grappling with but did so in a delightfully simple way, fully embracing the use of *attribute names*. We also discovered SQL but quickly rejected it as being not only unfaithful to the relational model but also extremely baroque and unconventional. And *relationally incomplete*. I assured the executive manager who was to approve our funding that SQL would never catch on!

4. Terminal Business System

No further notes—see Section 3, A Brief History of Me.

5. Report Writer Deficiencies

No further notes—see Section 3, A Brief History of Me.

6. How The RM Could Come to The Rescue

No further notes—see Section 3, A Brief History of Me.

7. Planning Business System 12

1. *What about "calculations"?*

   TBS's Report Writer supported the definition of "calculated fields", very much in the style of the additional attributes of **Tutorial D**'s EXTEND operator. We couldn't understand why Codd's model didn't support such a feature. When he was approached on this issue his response was that he regarded that as something to be addressed in the host language—remember that he used the term "data sublanguage" in his proposal, expecting such a language to be incorporated into existing programming languages, perhaps in the style of embedded SQL. This response was unacceptable to us because Report Writer's calculated

fields had names, like those of the input file, and could thus be referenced in subsequent commands.

ISBL's solution, what we now call extension, may seem obvious, but it hadn't occurred to me or my colleague in the planning of BS12 because we had been taught that Codd's algebra was relationally complete and we didn't consider that adding a new primitive operator might be a proper and acceptable thing to do! I'm now inclined to think that our lack of boldness on such matters might have been an advantage: Codd's model was sacrosanct to us and we didn't consider that we had any authority to deviate from it.

2. *How to handle duplicate attribute names arising from, e.g., equijoin? (Dot qualification clearly unsatisfactory)*

   It had not occurred to us that the so-called "natural" join operator could be taken as primitive without loss of what had taken as the more general *theta*-join, which was defined in terms of Cartesian product and restriction. But the Cartesian product of two relations whose headings are not disjoint appeared to produce a relation with two attributes of the same name. The only solution to this problem that we were aware of before we studied ISBL was the use of "dot qualification" in Codd's notation for his relational calculus (later emulated in SQL); but this solution was unacceptable to us. We assumed that the dot-qualified names would survive as attribute names in the resulting relation. If that result were then an operand to another relational operation requiring dot qualification, we would have doubly dot-qualified attribute names and repetitions of this process would produce longer and longer attribute names, threatening to exceed the rather small maximum name length (32 characters, if I remember right) that we were willing to support in those days.

   ISBL's neat solution—including the RENAME operator—gave us great relief and much joy. SQL's approach, by contrast, struck us as quite ridiculous (hence, my conviction at the time that SQL would "never catch on"!).

3. What does he really mean by "domain"?

   When Codd was asked if his "domain" concept was really just the familiar one we used to call "data type", his answer was an emphatic "no". In that case it had to be something unfamiliar to us, which was a bit scary. The ISBL folks decided to take it as meaning data type anyway and that gave us some relief—in those days we didn't have the understanding of user-defined types that we have now. But ISBL supported just a few simple built-in types and had next to nothing by way of support for integrity. We decided that "integrity" meant domains and a domain was a simple built-in type qualified by a constraint. Moreover, the constraint in question could access the database—for example, to check that the supplier number for a shipment was indeed that of an existing supplier. (We knew the term "foreign key" but we thought of it conceptually only and hadn't considered that the concept might be the basis of a convenient shorthand.)

   Sadly, the 1992 edition of the SQL international standard introduced CREATE DOMAIN, exactly repeating our error in BS12—this in spite of the fact that I was by then a member of the relevant international committee. Actually, I thought the feature was possibly useful anyway, knowing that "proper" UDT support was to come later (ha!), so I proposed to change the name rather than discard it altogether. Unfortunately, I couldn't think of a better alternative name than something like "generic column definition", so my proposal was rejected. Later, all the key players in the standard came to regret the inclusion of CREATE DOMAIN and few, if any of them implemented it. Also, SQL is guilty of extreme lack of orthogonality with respect to these domains: they cannot be specified wherever a data type can be specified.

8. Some General Language Features

The abbreviation rule was inherited from TBS. Although menu-driven end-user interfaces were becoming voguish, we expected fairly heavy use of command lines, especially by DBAs, for ad hoc purposes. *Caveat lector:* In subsequent notes I randomly lapse into use of these abbreviations in rather whimsical fashion. I hope this doesn't annoy you too much.

The use of prefix notation, as illustrated on the slide, gave us uniform syntax for operator invocations, both built-in operators and user-defined ones. At one time we were seriously considering that users could thereby "overload" built-in operators, but fortunately we came to our senses on this matter in good time. (Imagine having to write something like SYS.JOIN instead of just JOIN when you want to make sure the built-in operator of that name is invoked.)

9. Notable Features in Release 1

I mention these features because they are all ones that were missing from original SQL and didn't arrive in the SQL standard until 1996 or 1999. Our experiences with TBS allowed to anticipate many requirements that the people in research labs were unaware of.

That said, deferred constraint checking was a poor relation of what SQL now supports (and *TTM* rejects in favour of multiple assignment with checking at innermost statement boundaries). Deferment was by use of a BYPASS option on update statements, meaning "bypass domain checks" only—key constraints were not bypassable. You needed special authority to use BYPASS, which is just as well, as the deferment could be indefinite, even across outermost transaction boundaries! It was really intended to be used when possibly "unclean" data is being imported from another environment. A subsequent invocation of VALIDATE would report any violations of domain constraints and thus assist with the "cleaning" process.

Regarding triggers, we didn't have separate triggered procedures for INSERT, UPDATE, and DELETE. Rather, if a "stored table" (base relvar in *TTM* terms, base table in SQL) had an "update CLIST" defined for it (CLIST=command list, i.e., a proc!), then that procedure was invoked every time the stored table was the target of some update operation.

Regarding CREATE … LIKE …, I was very surprised, when I joined the SQL standards committee, to find that SQL's counterpart takes only a simple table name rather than an arbitrary table expression. I was unable to get this changed in the standard, even when CREATE TABLE <table name> AS <table subquery> was later added!

10. Scalar Types (= "System Domains")

Character strings were unlimited in length, subject only to available disk space. The quotes enclosing CHAR literals could be single or double, saving the need to "double up" inside the literal except in extreme circumstances. Counterparts of SQL's VARCHAR(*n*) were obtained using domains. At this point it is useful to mention the special userid INSTALL, whose database was populated and maintained by those responsible for the BS12 installation and was available on a read-only basis to all other users. Another userid, SYSTEM, was built-in and its database was also available to all on a read-only basis. The expression DOMAINS(SYSTEM), or DOM(SYS) if you prefer, denoted a table giving the definitions of the "system domains" shown on this slide. You might expect INSTALL to be so kind as to provide a domain named INTEGER, defined on NUM with no decimal places.

Values of type NAME were effectively character strings in which all alphabetic letters were in upper case, no space character appeared at the beginning or end, and no space character was immediately followed by another space character. A literal of this type was written as a quoted string immediately followed by N. The string in quotes was implicitly converted to the canonical

form I've just described, such that the comparison ' Ted Codd 'N = "TED CODD"N evaluated to '1'B (i.e., true). Columns of type NAME were used in the catalog tables for table names, column names and so on.

The curious BIT type for truth values was taken from IBM's assembler language for its System/370 computers. TBS had been written entirely in this language.

Timestamps were recorded as the number of chronons (centiseconds, I think) since or before the "beginning of time" (midnight on January 1st, 1900). Negative values represented times before the beginning of time, of course.

I don't know why type SYNTAX wasn't first-class and why it wasn't used for the relational expressions denoted by the values of column *c* in LIST(*t,c*) (see Slides 13 and 14). The type of *c* when LIST(*t,c*) was an operand of PRESENT (projection), was NAME, the values denoting column names.

11. Non-Scalar Types

As the slide makes clear, table types were not first-class in BS12. One reason for that was due to our (and everybody else's) understanding of first normal form—we thought there was a law against allowing tables to appear at row-column intersections!

Regarding table types for parameters, in fact the only permissible such type was the general one, TABLE, for all tables. Thus a view or function could be exposed to a run-time type error if it made an unsafe assumption about the specific type of an argument substituted for a parameter of type TABLE. On the other hand, BS12 installations were able to, and did, provide Codd's relational divide operator as a parameterised view, whose definition was expressed using join, semidifference, projection, etc.

The complete lack of explicit row types didn't matter much, thanks to the GENERATE operator mentioned on the next slide, as I explain in the next note.

It was Stephen Todd, my contact at the IBM scientific centre responsible for ISBL, who alerted me to the existence of columnless tables, but alas it was rather late in the development cycle for BS12 that he did so. I remember his email to me, saying something like, "Hugh, I forgot to tell you about tables with no columns …", and the joy I experienced at such a cute but logically impeccable concept. I immediately told all my colleagues, pleading with them to do what would be necessary to allow for such tables in the components they were responsible for, but alas not all of them could promise to make the required changes within our agreed deadline for delivery of Release 1. (The names TABLE_DEE and TABLE_DUM occurred to me many years later, by the way, when I was deliberately writing about my distaste for SQL in a rather zany fashion.)

12. "The 12" Relational Operators

These are what gave BS12 its name. Apart from GENERATE, EXCLUSION, and MERGE, they were all adapted from direct ISBL counterparts, though we added some embellishments, such as EXCLUDE in PRESENT and the LIST construct in various places.

CALCULATE was our counterpart of ISBL's rectification of that perceived deficiency in Codd's algebra. **Tutorial D**'s counterpart is EXTEND. The name given to a calculated column could be accompanied by a domain name, in which case that would be the domain of the calculated column and system would check that the result of each calculation satisfied the relevant domain constraint.

GENERATE was effectively CALCULATE without a table operand—nowadays I would say it implicitly extended TABLE_DEE. It was most commonly used to write literals denoting one-row tables. Thus **Tutorial D**'s REL{TUP{X 3, Y 4}, TUP{X 5, Y 6}, …} could be written as

UNI(GEN(X = 3, Y = 4), GEN(X = 5, Y = 6), …), which arguably had the advantage of clarifying how duplicate rows would be handled!

The names INTERSECT and DIFFERENCE for semijoin and semidifference must raise some eyebrows and I apologise for them now. They arose because the ISBL folks described their counterparts of those operators as generalisations of Codd's intersection and difference. That being the case, we rather naively thought that the generalisations could take the same names.

MERGE is the operator I described in *Relational Database Writings 1989-1991,* Chapter 20, "Outer Join with No Nulls and Fewer Tears". The third operand, *t3*, if given, whose heading had to be disjoint with that of *t1,* denoted a one-row table providing the row to be joined with each row of *t1* that has no matching row in *t2*. The columns of *t1* were required to include common columns corresponding to the key of *t2*, thus ensuring that the join was *n*-to-1 from *t1* to *t2*. That didn't mean that *t2* had to denote a stored table, for in BS12 every table had a key, as I explain later (Slide 20). The reason for the *n*-to-1 restriction was that it seemed arbitrary and suspect to generate, as SQL's outer joins do, exactly one result row for an unmatched *t1* row if *t1* rows in general could match several *t2* rows. It also allowed us to support update-through-MERGE (see Slide 18) without the possibility of violating the Assignment Principle as that principle must apply to a system that has no concept of "cascaded" update operations.

The "big mistake" noted on the slide is explained as follows. QUAD required its operands to have no common columns, just like **Tutorial D**'s TIMES. However, unlike **Tutorial D**'s JOIN, according to the reference manual BS12's JOIN required its operands to have at least one common column. Now, JOIN(*t1,t2,t3*) was defined as equivalent to JOIN(JOIN(*t1,t2*),*t3*). If the reference manual is right, this means that, for example, JOIN(S,SP,P) works but JOIN(S,P,SP) doesn't because S and P have no common columns. Now, we were well aware that QUAD was really a special case of JOIN but we felt a need to protect users from inadvertently asking for Cartesian products that might produce very large tables indeed. If they were forced to write QUAD for such cases then we would know they really meant it. The observation that the associativity of JOIN was thus spoiled was made to me long before we first released BS12 and I thought we had addressed the problem by generating a warning message for joins with no common columns, but the reference manual, if it is correct (it might not be!), tells me that my memory is wrong in that regard.

13. BS12's Projection/Rename Operator

Note the counterpart of **Tutorial D**'s ALL BUT construct, EXCLUDE(*column-name-list*). The ALL option was needed for the case corresponding to **Tutorial D**'s RENAME operator: PRESENT(T, RENAME(X,Y)) was equivalent to T{X} RENAME (X AS Y), so to achieve the effect of T RENAME (X AS Y) you had to write PRE(T, ALL, REN(X,Y)). The NONE option was added hastily when columnless tables came to our attention. Alas, such tables could not be final results for transmission to clients, nor was a stored table or view permitted to have no columns, in spite of my pleading ("Oh, come on, guys!") with colleagues who were responsible for these areas.

14. Other Uses of LIST(*t,c*)

When LIST(*t,c*) was used to provide the operands of *n*-adic relational operators, the type of the column *c* had to be CHAR. It seems as if it should have been SYNTAX, given that the values were table expressions, but the reference manual clearly states that type SYNTAX was available only for parameters. I think the reason was that the idea for type SYNTAX—indeed, the idea that BS12 would use "call by substitution" rather than "call by value"—came about rather late in the development cycle, like columnless tables.

15. Other Operators of Type TABLE

The main point of this slide is to illustrate BS12's slavish adherence to what I later learned as Codd's *Information Principle*—which we now prefer to call *The Principle of Uniformity of Representation*.

We decided to use operator names rather than variable names for all these "system tables" to save users from having to use dot qualification to avoid possible clashes with their own table names.

Note the various "tables of tables": TABLES(OWN) for the current user's stored tables and permanent views, TABLES(INSTALL) for those belonging to the special userid INSTALL mentioned in the note for Slide 10, and so on.

In addition to those shown on the slide, BS12 had:

- The "cumulative accounting table", TACCOUNT(), effectively telling the system administrator how much each user's previous use of the system in the "current accounting period" was going to cost them.
- The "session accounting table", SACCOUNT(), giving accounting information for each completed user session in the current accounting period.
- The "table access wait status table", QUERY(WAIT), telling users which stored tables (including view definitions) they were waiting for access to on account of locks held by another user, and how long they had been waiting for it in each case.
- The "PU status table", QUERY(PU), containing "information on processing unit consumption"—paying customers were charged by the processing unit, whatever that was.
- The "table of user profiles", SYSTEM(PROFILE), containing a single row giving the invoking user's profile when invoked by an ordinary user, but giving the profiles of all users when invoked by the system administrator.

16. The DEFINE Command

This was the most commonly used command in BS12. All sorts of advantages came from being able to assign a name to a table expression, that name to stay in scope until the end of the session. I wanted this in **Tutorial D** too but alas the burden of defining exactly what a session is was not worth the hassle, considering the purpose of **Tutorial D**, and in the end we made do with WITH (which is applied rather more generally than it is in SQL).

The special name * denoted the "current table" (if any). Note that DEFINE was the "default command", meaning that its key word, DEFINE, could be, and usually was, omitted altogether. Moreover, if the key word was omitted, then "*table-name* =" could also be omitted, such that a table expression *x*, given where a command was expected, was short for DEFINE * = *x*.

An invocation of START TRANSFER (see Slide 23) also caused the current table to be (re)defined, as that denoted by the *table-exp* operand.

17. Updating

The "single row" commands, ADD, CHANGE, and DELETE, were really inherited from TBS. They are redundant, of course, but we thought it useful for the user to clarify the intent to affect just one row, to avoid certain accidents like deleting rather more than one intended.

Note the lack of WHERE clauses on UPDATE and CLEAR. They weren't needed because you would simply use an appropriate SELECT invocation as the target:

```
UPDATE SEL(Emp, Dept# = 'D5'), Salary = Salary * 1.1
```

Note the USING option of UPDATE, where the USING operand was effectively a table representing transactions to be processed against the target table. Thus we emulated the concept of "batch processing" that had been the usual way of updating databases in the 1960s and 70s. **Tutorial D** doesn't have this but perhaps an Industrial D should. (How do SQL users get the same effect?)

18. Updating through *table-exp*

BS12's restrictions on view updating might seem very conservative, especially in the light of Chris Date's current proposals. We didn't have a name for *The Assignment Principle* in those days, but we were certainly guided by it. This was not based on advice we got from anybody else. We just thought that any invocation of an update operator should have precisely the effect on the target that was defined for that operator. We also thought there should be no side-effects other than those explicitly defined in "update CLISTs", BS12's counterpart of SQL's triggered procedures.

We could perhaps have relaxed the key-preservation requirement on inserts but note that SQL's WITH CHECK OPTION was always in effect and could not be overridden, even by use of a BYPASS option.

Note also that the rules for PRESENT and MERGE required the keys of the operands to be known to the system—see Slide 20.

Projection (PRESENT)

> On insert (ADD and STORE), the default value for an unassigned column was taken from the definition of the domain for that column. The system domains (built-in types) had system-defined defaults, of course: 0 for NUM, " for CHAR and "N for NAME, and the beginning of time (midnight on January 1st, 1900) for TIMESTAMP. The reference manual doesn't seem to give the default value for type BIT.
>
> The requirement for the projection to preserve the key of its operand meant that deleting *n* tuples from the target also deleted *n* tuples, and no more, from that operand. A similar comment applies to CHANGE and UPDATE. Insert through non-key preserving projection might have been supported but we decided it was not a good idea to allow key columns to be assigned default values.

Restriction (SELECT)

> All updates were propagated to the operand table. Each inserted or updated row was rechecked for satisfying the restriction condition, the update being rejected if the check failed.

Extension (CALCULATE)

> All updates were propagated to the operand table. The reference manual is unclear on the treatment of calculated columns and the question arises in my own mind now: what happens if an update causes some divisor to become zero in the expression defining a calculated column? It seems reasonable to assume that the extension expressions were all re-evaluated so that the update can be rejected if necessary.

Semijoin (INTERSECT)

> All updates were propagated to the first operand table. Each inserted or updated row was rechecked for existence of a matching row in the second operand, the update being rejected if the check failed.

Semidifference (DIFFERENCE)

All updates were propagated to the first operand table. Each inserted or updated row was rechecked for *non*existence of a matching row in the second operand, the update being rejected if the check failed.

Left outer join (triadic MERGE)

All updates were propagated to the first operand table.

Special case join (dyadic MERGE)

The treatment was as for semijoin.

The Table of Tables (TABLES(OWN))

An insert caused the creation of a new stored table or view, *T*. Unfortunately, stored tables were required to have at least one column and a nonempty key, so *T* was flagged as unusable until at least one row appeared in COLUMNS(*T*). In the case of a view, the value of COLUMNS(*T*) was determined by that of DEFINITION(*T*), so that table would have to be populated before *T* became usable.

Assignments in updates via CHANGE and UPDATE were permitted only for certain specified columns, as you might expect.

Deletion of a row for stored table *T* had the same effect as DROP TABLE *T* in SQL. The deletion would of course "cascade" to COL(*T*) and DEF(*T*) as necessary. A popular demo trick was to give the command CLE TAB(OWN), then show the audience how that had conveniently destroyed the entire database. The demonstrator would then say something like, "Actually, I didn't really mean to do that because I have some more features to show you in this demo; so I'll just get it back for you." The next command would then be CAN UNI (cancel unit, i.e., rollback), given with a flourish and a broad grin. (Both commands ran at the speed of light, of course, because the hard work for destroying the database would have been done in response to END UNIT.)

The Table of Columns (COLUMNS(t))

BS12's counterparts of SQL's ALTER TABLE *t* were normal DML commands against the table of columns for *t*. In Release 1 it was not possible to add or drop a column once the first update on *t* had been performed. It was that first update that caused the system to commit to a storage structure for the table.

Language tables (DEFINITION(t))

Nothing special here. Each row in a language table defining a view, function, clist, or procedure consisted of a line number and some text and normal DML commands were used. General-purpose front-ends such as the one I eventually developed myself ("Dbird") were expected to provide "full-screen" editors for these tables. (This was before the advent of windows, of course.) In fact all other "system tables" were updated using regular DML …

The Table of Outstanding Messages (SYSTEM(MESSAGE))

… except for this one. It appears that this particular table was not updatable using regular DML. A MESSAGE command was available for sending messages to other users (or possibly oneself) of the same database.

The Table of User Profiles (*SYSTEM(PROFILE))*

For an ordinary user this was a one-row table on which insertions and deletions were not available. For the system administrator (INSTALL) this contained all the user profiles and normal DML was used to create new users and destroy departing ones. Of course adding

and deleting rows to and from this table had numerous side-effects on other system tables, as you would expect.

Other system tables

The other tables mentioned in the notes for Slide 15 were all entirely system-maintained. Internally, of course, adding, deleting, and updating rows were via the same methods as for other tables.

19. Integrity

BS12's support for integrity is pretty embarrassing by modern standards—better than SQL was at the time but SQL overtook it in 1989 when the so-called "referential integrity" feature was added to the international standard.

The only declarable constraints were key constraints (just one for each stored table) and domain constraints. It is true that a domain constraint could include table expressions referencing the database, so the domain for supplier numbers in the shipment table SP could specify exactly those supplier numbers that appear in the suppliers table S, but in that case SNO in S and SNO in SP were defined on different domains. Now, you might think that this was not necessary, because the SNO values appearing in S would always be exactly those values! But alas, the checking of domain constraints was performed only on values presented for input to the database, just as such checks had always been done in the old days. It is very upsetting that to this day students (and others) still have to be taught that enforcing database consistency requires rather more than just checking input values.

The BYPASS option offered a modicum of deferred constraint checking, but deferment meant indefinitely, even across outermost transaction boundaries! The VALIDATE command was available for discovering inconsistencies, but these would subsequently have to be resolved "manually". My suspect memory tells me that use of BYPASS caused tables to be flagged as "possibly inconsistent", but I can find nothing in the reference manual to corroborate that memory.

20. Keys

The slide pretty well shows how crude the heuristics were for computing keys of derived tables. We didn't get anywhere near doing a full analysis of functional dependencies along the lines of what is now specified in the SQL standard (though probably not much implemented yet). This failure was partly due to an error of omission in the original architecture for BS12 (by me, alas!—see the notes for Slide 27).

Regarding the failure to support multiple keys, I think we just thought that there was no real requirement as "nobody had ever asked for this" as a TBS user. It's possible, though, that Codd's emphasis on *primary* key, which we now think to have been much too strong, contributed. It's interesting, too, that ISBL—or at least the implementation, PRTV, of ISBL—had no support for keys, in spite of the fact that it also had no "support" for duplicate rows! Similarly, original SQL had no support for keys—and it, of course, *did* "support" duplicate rows. So perhaps BS12 was at least ahead of the field at the time, thanks to its predecessor, TBS. What's more, in bothering to attempt to compute keys for derived tables it probably remains ahead of the field, albeit in its grave.

I leave it as an exercise for the reader to spot the various deficiencies in the rules given on the slide.

21. CREATE Options

CONTENT

In addition to the options shown on the slide, the content could be something else as specified by the user. In that case, the treatment by the system was the same as for DATA but applications could make use of it for their own purposes. For example, one general-purpose front-end used CONTENT(EXEC) for tables containing procedures written in its own scripting language (and these procedures were allowed to include BS12 commands).

In the case of language tables, it should be noted that when one was "executed" its content would have to processed in the obvious order: LINENO ASC. There was quite some discussion concerning the legitimacy of using tables for program code, considering the relational rule outlawing any significance to any order in which the tuples might appear. We decided that we were not really out of line here, because the ordering used for execution purposes had no bearing at all on the relational operators as applied to these tables (not that much use of such operators was to be expected, as you can imagine—SELECT, for applying quick corrections to individual "lines", perhaps, and EXCLUSION for comparing two copies or versions of the same chunk of code to discover the differences, if any).

LIKE (table-exp)

When I joined the SQL standards committee I was horrified to see that SQL's counterpart of CREATE … LIKE … required the LIKE operand to be a simple table name, not an arbitrary table expression. My attempt to fix that problem was rejected and the problem remains to this day. Curiously, the same problem does not arise with CREATE TABLE … AS <subquery>, which appeared in SQL:1999. BS12 did not have a shorthand for simultaneous creation and population. We wrote CRE *table-name* LIK *table exp* followed by STO *table-exp* INT(table-name) (no, I don't suppose anybody really bothered to abbreviate LIKE, STORE, and INTO, but for some reason I keep liking to remind you of that idiosyncratic abbreviation rule—sorry about that!). And if *table-exp* was too cumbersome to type out twice in those days before copy-and-paste, then the DEFINE command came to the rescue.

SESSION

The created table was automatically destroyed at logoff.

REPLACE

If a stored table of the given name already existed for that user, then it was destroyed; otherwise the option had no effect, thus avoiding unwanted error or warning messages that would arise on explicit attempts to destroy a stored table that didn't exist. I know of no counterpart in SQL for this option, which might be a suitable candidate for inclusion in an industrial *D*.

KEY(column-spec-commalist) and COLUMNS(column-spec-list)

A *column-spec* was a column name followed by a domain name. (Recall that the built-in scalar types were called system domains.) I see now that this syntax, whereby key columns were defined inside the KEY specification, could not easily be extended to support multiple keys, considering that a relvar attribute can be a member of more than one key.

SERIAL(column-name)

This option was an alternative to KEY(…). A table created with this option was called a serial table. The specified column constituted the key and has its values automatically generated by the system when rows were inserted. The domain of this column was NUMERIC but its values were integers—BS12, having no built-in integer type, allowed

users to defined their own but was restricted to the system domains for cases like this. A similar problem arose with the LINENO columns of language tables.

By the way, serial numbers started at 1 and increased by 1, in the order in which rows were added to the table. The STORE command had an ORDER option in case the user wanted some control of the assignment of such serial numbers. Being a member (the only member) of the key, the column was not assignable to in CHANGE and UPDATE. Also, although rows could be deleted from such a table, their serial numbers would not be reused on subsequent insertions.

Support for serial tables was included mainly because TBS had support for something similar: files, with fixed-length records, whose records were access by record number rather by key values. Our implementation of serial tables also matched TBS in a certain respect: the serial column occupied no space on disk! (See Slides 31 and 32 for further implementation details.)

ORDER(order-spec)

The specified order was applied by default on transfer of data to the client. The *order-spec* was the usual list of column names with ASC or DESC qualifiers (and we didn't think much of original SQL's use of column numbers instead of names—now withdrawn from the SQL standard). Where possible, the default ordering was preserved through relational operations.

22. Language Tables

See the notes for Slide 22. There's not much more I can say about these. Note that a view name *v* generally referred to the table defined by *v*, as in SQL and as with virtual relvars in **Tutorial D**. The obvious exception was DEF(*v*), which denoted the language table containing the text defining *v*.

23. Data Transfer

START TRANSFER was BS12's counterpart of SQL's OPEN CURSOR. Notable differences:

- No name was assigned to the "transfer session" initiated by use of this command. This was because no more than one transfer session was permitted concurrently per user session. The reason for this (highly controversial!) restriction was nothing to do with ease of implementation. Rather, we wanted to do our level best to prevent client applications from doing what is better done by the system (e.g., joins). Of course *TTM* outlaws cursors altogether (RM Proscription 7) for much the same reason.
- The specified table became the "current table", denoted by *, replacing any existing definition of *.
- A transfer session could be ended by END TRANSFER, meaning the same as CLOSE in SQL, or by CANCEL TRANSFER, in which case any updates to * during the transfer session were undone.
- BS12's counterpart of FETCH was GET, which caused a row to be the current row if the transfer was for both retrieval and update. In that case subsequent CHANGE commands with * as target were taken to refer to the current row, to allow specified columns in that row to be reassigned. PUT would then replace that row in the database.
- According to the reference manual, PUT could also be used to insert a new row into the current table, but the manual does not explain how the contents of this row were established and I've forgotten the details.

- A RETURN option was available for asking the new or updated row to be returned to the client.
- The FORMAT option was used to specify, for each column, the format in which data was to be transferred between client and server. If no format was specified for a particular column, then a default format was taken from the Table of Columns for the current table, COL(*). If the column in question was not directly derived from a column of a stored table, then the default format was taken from the SYSTEM(PROFILE) row for the user, which provided default formats for each built-in scalar type.

  Actually, there were two default formats for a column, termed the external format and the internal format. The external format would normally specify some conversion to character strings, suitable for use in visual renderings; the internal one might use representations more suitable for computer programs, such as 32-bit integer or an IEEE floating-point representation. Character strings could be presented as fixed length strings with blank padding or as varying length strings, the latter being a little more awkward for programs in those days.

  Regarding the default formats in SYS(PRO), the reference manual includes the intriguing text: "[W]hen no format is provided in the SYSTEM(PROFILE) table …". As BS12 had nothing like SQL's NULL, I'm not sure what that meant! Anyway, it goes on to specify the ultimate "system" default formats for each data type. I am most amused, now, to see that the default format for TIMESTAMP values was D'DD/MM/YY' T'HH:MM:SS:TT', in spite of the fact that dates outside of the current century were supported. Note also the lack of American influence on this default format—not surprising when you consider that IBM Corp. in the USA was barred from offering a bureau service.

  A somewhat amusing little problem arose out of our extensive support for formatting alternatives. Some of these options involved truncation or rounding, and this could give rise to what looked like duplicate rows in displays and print-outs!

24. Implementation

My notes here are entirely from memory, as the reference manuals of course have nothing to say on this subject.

25. Time Line and People

Two of those five planners in 1978—myself and a colleague—were responsible for the initial specification. For a reason I never understood, the lead planner, who had been campaigning for a new product for the bureau service for quite some time before 1978, had in mind the notion of what he called an "advanced end-user facility". I had been the one to suggest a relational DBMS and I had tried hard to educate this leader concerning the fact that a DBMS would have to be a server and thus could hardly be called an end-user facility. Nevertheless, we were stuck with the provisional and awful name AEUF for a long time. The name Business System 12 eventually emerged when some of the language features were pinned down (see the appropriately numbered Slide 12).

My colleague and I had a jolly time wandering around various research or quasi-research departments within IBM to find out what progress had been made regarding implementation of what Chris Date had taught me in 1972. We struck gold when we visited IBM UK's Scientific Centre in Peterlee, where the language ISBL had been developed and implemented as the Peterlee Relational Test Vehicle (PRTV), so named because IBM didn't want it to have a glamorous name that people might take as an intended successor to IMS and DL/1. (Peterlee is in the north-east of

England, not far from Newcastle where the *TTM* Implementers' Workshop took place. (A funny feeling came over me as I drove past the road signs to Peterlee on the A1 on my way to Newcastle.)

For the purpose of demonstrating our ideas, I developed a prototype of the query language (the 12 relational operators) using a scripting language in VM/CMS called EXEC2. This was regarded as a *tour de force* by those acquainted with EXEC2 (whose deficiencies later inspired IBM UK's Mike Cowlishaw to develop Rexx, my favourite personal computing language which I still use, in Windows, to this day).

When the planning was complete and funding had been obtained for the development, I shifted from the planning department to the development department and took the lead on the bits I was qualified to take on as a result of my experience on TBS: data storage and retrieval, including query evaluation ("execution of the relational stuff", as it says on the slide), which explains why those aspects are the only ones I have anything to say on in this presentation.

High-level language design at that time was absolutely not something I, an assembler language programmer by trade, was into and I was deeply impressed by what my colleagues in that area had made of the ideas we had obtained from ISBL, combined with the generally perceived "database-y" requirements that had previously been addressed in TBS. One of those colleagues had previously developed compilers for Fortran and PL/I for the benefit of users of the Bureau Service's proprietary "personal computing" environment, VSPC (see next slide). It was he who was primarily responsible for BS12's programming language—our counterpart of what was to become SQL/PSM in SQL:1996 (or PL/SQL in Oracle).

Regarding Dbird, I still have the user guide I wrote for it, dated September 1984 and printed on continuous flow stationery. It runs to 85 pages, including the index. Its features, as listed on the table of contents, were:

- DIY mode—essentially a command line interface, with commands like DISPLAY *table-exp,* EDIT *table-exp* (for language tables), EXECUTE *table-exp* for executing Dbird procedures, not BS12 ones, REPORT *table-exp.*
- Dbird reports. This feature was my "dream coming true", with reference to that epiphany I had when attending Chris Date's course back in 1972. The various features of the scripting language I came up with were all to do with report layout and what we used to call "control breaks" (perhaps that term is still used): exploiting the ordering specified for the rows to determine which column values get printed for every row, which when the department number changes (e.g.), which when the branch number changes, which when the area code changes, and which at the very end ("grand totals"). All the hard work of producing the data for the report, including "totals" (results of aggregation, via SUMMARY), was done by BS12, of course.
- The Table Displayer, which supported two formats: column-wise and row-at-at-time (with column names down the left margin), switchable on the fly.
- Assisted Query Building, menu driven support including menu items corresponding to each of the 12 relational operators, allowing selection from lists (e.g., of tables or columns) where appropriate.
- The Program Editor for functions, CLISTs, procs and view definitions (though views could also be developed in Assisted Query Building, via BS12's KEEP command).
- Assisted Table Creation
- Full-Screen Data Entry
- Importing and Exporting
- Interfaces for PL/I programs, including the ability to invoke Dbird from such a program.

26. Environment

No further comments.

27. Gross Architecture

This slide illustrates the Chief Architect's annoying but uncontrollable habit of inflicting his favourite hobby on all and sundry at every opportunity. The two single-letter components, X and N, don't follow the style of the others, but that's because they were added later by other people—in their defence, they did point out that eggs (X!) and nests are also ornithologically related. The architecture was the easy bit. Naming the components wasn't so easy: they had to be bird names such that no two began with the same letter (to satisfy the module naming convention: every module name was 8 characters beginning with the IBM-wide unique prefix I had managed to obtain, BRD, followed by the first letter of the component, the remaining four being chosen arbitrarily). Also, each had to be such that it could be treated as an acronym for a phrase describing the component in question. Sorry that I can't now remember exactly the phrases that Scaup and Pipit stood for. Osprey was an exception. I was unable to think of a bird beginning with O that could also be a suitable acronym for the optimizer, so I just made do with my favourite bird beginning with that letter.

TEAL

> This was the root component, so to speak. It was responsible for parsing input commands, checking for existence of referenced objects, type checking, transforming into a form suitable for processing, and the basic algorithms, sans optimization, to implement the semantics. I believe Teal also included all the code for the programming language used in user-defined functions etc. (if not, then there was another component that should appear on the slide but even if I could remember it there wouldn't be room for it on the slide).
>
> For example, the output of parsing and checking a START TRANSFER command would be a pipelining tree (see next slide) for evaluation of the table expression denoting the table whose rows were to be transferred to or from the client. That tree might include dyadic JOIN node, connected downwards to two nodes representing the operands of the join. The program, BRDTJOIN, "residing" at that node, effectively implemented the simple nested loop algorithm, leaving it to Osprey to inject extra nodes into the tree to make that algorithm go faster. X was really a subcomponent of Teal to handle scalar expressions. When these appeared as conditions in SELECT invocations or on the right-hand side of assignments in CALCULATE and SUMMARY invocations, they gave rise to trees hanging off the relevant nodes in the pipelining tree and these X trees were known as epiphytes.
>
> My own contribution to Teal, as a coder, was the programs like BRDTJOIN, implementing the semantics of the relational operators.

GULL

> Gull's programs resided at leaf nodes in the pipelining tree, representing stored tables. Gull understood tables and decided how best to store and access them on disk using the facilities of Lark. Gull was developed by a single programmer (me), who also wrote the spec for Lark and Pipit. For further details see Slide 32.

OSPREY

> Osprey, the optimizer, understood the pipelining tree and the semantics of relational operators very well. It also understood keys. It was initially developed by a single programmer (me) but a second programmer joined in at about the halfway mark (and a most

enjoyable collaboration that turned out to be—optimizers are hairy beasts and ours was no exception!). For further details see Slide 29.

LARK

Lark knew nothing about tables. It provided services for creation and access of things on disk called lists. A list was a collection of records of the same length, either a sequential list or a hash list in Release 1 of BS12. The records of a sequential list were stored in an order such that each record could be accessed by its record number; those of a hash list were stored at locations determined by hashing their content, and in this case no two records were permitted to have the same content. Lark used Pipit to obtain (on demand) and subsequently access pages on disk in which the lists were stored. Lark was also the responsibility of a single programmer, though a second programmer was involved in the support for B-trees added in Release 2. Further details on Slide 31.

PIPIT

Pipit knew nothing about lists but all about the pages of a VSAM file on disk. Lark was probably the only user of Pipit. Lark knew about the files managed by Pipit. Pipit knew which pages in those files were currently unoccupied and therefore available on request. Pipit also knew where to find a particular page when asked for it—a copy of it might already be in what we now call RAM or, if not, Pipit would create such a copy by reading from disk. Use counts, and probably a when-last-used timestamp, were associated with each page so that Pipit could make a sensible decision when it was necessary to overwrite an existing copy in RAM.

SCAUP

Scaup was available to all other components for resource (memory, CPU) allocation. It also provided locking services, supporting read locks, write locks, and the new idea at the time, intention locks. These locks were used for serialising access to the database, when necessary, held until the outermost unit was ended or cancelled. In Release 1 they were locks on stored tables but the granularity was improved to row-level locking in Release 2. We thought it very important for database locks to be on objects known about by users, because users are inevitably aware of their effects. For that reason we rejected the idea of using, for example, locks on pages. Scaup detected deadlocks but I can't remember how they were resolved.

There were also short-term locks—"spin" locks—for internal use in circumstances where releasing the lock did not depend on action by the client at some indeterminate time in the future.

Scaup also handled all data transfer between server and client.

28. The Pipelining Tree

The interface supported by the programs residing at each node was effectively a propagation of the BS12 commands used within a transfer session (see the notes for Slide 23). For example, when the JOIN program, BRDTJOIN, received a GET request, it would determine whether a "current" row existed at its left operand node and if necessary issue a GET to that node and a RESET to its right operand node. Then it would issue a GET to the right operand. If that GET succeeded, it would ask X to evaluate the epiphyte at its own node, representing the join condition (comparing common column values). If the evaluation yielded a matching row, then BRDTJOIN returned to its caller (residing at the node above) with "success—I have a row for you"; otherwise it would issue another GET to the right operand, and so on, the outer loop ending when GET to the left operand resulted in "no more rows".

29. The Optimizer

The join process described in the note for Slide 28 was very slow, as you can see, but before any coding has started on Osprey we reached a stage in the development of BS12 where we had the pipelining tree delivering correct results (except that duplicate rows could appear) for table expressions of arbitrary complexity. "Have faith!" I entreated my boss when he winced at the dreadful performance at that time, secretly crossing my fingers, hoping that my plans for the optimizer would achieve what we expected of it. I'll take join as an example of the kinds of things it did.

First, a key was computed at each non-leaf node, based on the rules given on Slide 20, those at leaf nodes for stored tables having been placed by Gull. By examining the keys of the join operands, Osprey was able to determine whether the join was many-to-many, many-to-one, one-to-many, or one-to-one (from left to right). If possible, it ensured that a "one" node was on the right, switching the operand nodes if necessary. When the right operand was indeed a "one" node, it further looked to see if the key at that node corresponded to the key of a stored table, as it would if the node were a leaf node or a node with a direct line to a leaf node that preserved that leaf node's key. In that case, a GET request to the right operand could be satisfied by taking the current common column values and asking Lark if a matching record existed in the hash list for that key. An "access-by-key" node would then be inserted between the join node and its right operand. Now, when a GET to the access-by-key node was successful, if another GET was subsequently given to it without an intervening RESET, then it would return "no more rows" without further ado, thus saving the overhead of an expensive "search-not-found". Thus, joins that could be characterised as "join on foreign key" went very fast indeed.

Fore more difficult joins, Osprey would still insert some kind of "go faster" node, perhaps to construct a hash list on the fly such that the first iteration of the nested loop was exhaustive but subsequent ones used the access-by-key method.

30. Optimizing Duprem

The fact that in most cases duplicates can pass safely through a node in the pipelining tree was given to us by the ISBL people and it came as a great relief.

The need to determine whether a projection preserves the key of its operand was the prime motivation for determining keys of intermediate results (see Slide 20). I had had to fight tooth and nail to preserve BS12's "slavish adherence" against those who thought this would be infeasible for performance reasons and I would ultimately have lost that fight if those people had turned out to be right in spite of my best efforts.

31. LARK Features

See the notes for Slide 27.

I'm not sure about N, the transaction manager. I've previously mentioned that for each user there was a single VSAM file for the database and another for temporary stuff that lasted only for the duration of a session. It doesn't seem a good idea to include the log in either of these files because logs are used in recovery from accidental destruction, to restore updates made since the most recent backup. There might have been a third VSAM file, also managed by Lark, or N might have done its own thing (which I somehow doubt).

32. GULL Features

See the notes for Slide 27.

Vertical decomposition

The lists for a stored table were a hash list for the key columns and zero or more sequential lists for the nonkey columns. Gull chose element lengths for lists such that there would always be a certain number of elements (I forget how many) per page. When creating the list structure for a stored table, it would process the nonkey columns in the order in which it saw them, such that several columns were stored in each list. When the desired maximum element length had been reached, Gull would allocate a new one for the remaining columns.

My idea was that "smart" reorganisation could take note of which columns were typically needed in combination in queries and revise the structure accordingly. As I recall, we did collect the required statistics, but we never got around to implementing smart reorg (Release 2 was developed after I had returned to the UK).

The Literal Pool for long character strings

The literal pool was a single list for the entire database (i.e., there was one per user). A string was recorded as a length indication followed by its content. Strings that could be fitted into 12 bytes were just stored in the list to which the column containing them was allocated. Longer ones had their length and the first few characters stored there, accompanied by a hash token pointing to an element in the literal pool where the remaining characters could be found. If the remaining characters could not be fitted into a single list element, then the chaining process would just be repeated until the whole string was stored. If the same value appeared more than once in the database, only its first chunk was actually stored more than once—if the remainder already existed in the literal pool, Lark would allow that fact to be discovered and the existing hash token used as a pointer.

Note that the comparison $x = y$, where x and y are columns of a stored table (not necessarily the same table), could be evaluated without retrieving the entire strings from disk—in fact, without access to the literal pool. The two values were equal if and only if their initial 12-byte entries were equal and otherwise they were not, thanks to the fact that the literal pool, being a hash list, contained no duplicate entries.

"Identify"

A GET request to a Gull leaf node did not retrieve any data. It just resulted in a return code indicating whether or not a row was identified to satisfy that request. Expression evaluation at higher nodes would issue a "realise" request to Gull when encountering a column reference with no associated value. Moreover, the caller could specify, in the case of character strings, whether or not it could make do, *pro tem,* with just the first few characters. We thought this was an ingenious idea but alas it didn't seem to give the performance benefits we expected—I forget the reasons.

33. Lots More Could Be Said …

For example, my presentation is silent on the subject of nulls. Let me take the opportunity, here, to admit, with a blush, that initially I had been assuming we would allow a row/column intersection to be occupied by something we called a null. But it never occurred to me that the comparison null=null might result in something other than TRUE, nor that null=x where x is something other than null, would yield anything but FALSE. I changed my mind about nulls in the light of advice from several quarters. Our System/R contact (Bruce Lindsay) told us what an enormous headache they had caused in their SQL implementation. Stephen Todd, my ISBL contact, explained to me the "need" for three-valued logic in connection with nulls, and that scared the living daylights out of me—not because I could at that time envisage all the problems with 3VL that Chris Date has written about since then, but simply because I didn't feel we could thrust such an unusual, esoteric, *avant garde* idea (as I saw it) at our users, who would be fully accustomed to the rules of regular

logic. Chris Date also warned us off nulls, advocating instead his original "default values" proposal, and that was the approach we adopted. It wasn't so much of a struggle to convince others of the wisdom of this change of mind, as it had been to convince them that BS12 would have no notion of duplicate rows.

Of course the "default values" approach wasn't entirely satisfactory. Our very first BS12 customer was a company offering investment advice based on financial information provided by companies. They needed about 300 columns for one of their stored tables, which would have been nearly that number of tables if they had followed the decomposition approach. Most of these columns were numerical, where zero meant what it said and so could not represent a missing value. But the default value had to be one in the domain of the column in question, so values like -999999.99 pervaded their database and as a consequence they had to take great care over their queries, as you can imagine. The one gratifying consequence was that they, who had never previously done any computer programming, cheerfully studied our programming language and after we had coded up a couple of user-defined functions for them to prevent those default values from giving rise to misleading results, they set about coding the rest themselves.

My presentation on June 2nd, 2011 eventually came to an end with "Oh, I've just noticed that I've overrun my allotted time by 30 minutes. Sorry about that. I'd better stop right away." If I've overrun my allotted space too, sorry again!

The End