# Konkurrierende Dateizugriffe

In den vorangegangenen Kapiteln wurde bereits ausführlich darauf eingegangen, wie in COBOL Dateien definiert und verarbeitet werden. Neben der rein sequenziellen Dateiverarbeitung wurde auch insbesondere die Index-sequenzielle Organisationsform erklärt, der in Multiuserumgebungen eine besondere Rolle zukommt. Dieses Kapitel beschäftigt sich mit der Frage, wie Dateien zu definieren und zu verarbeiten sind, wenn sie gleichzeitig von mehreren Anwendern benutzt werden sollen.

Dabei lassen sich grundsätzlich drei verschiedene Vorgehensweisen unterscheiden: Verarbeitung ohne Satzsperren, pessimistische Satzsperren und optimistische Satzsperren. Um die Unterschiede aufzuzeigen, werden sie im Folgenden zusammen mit immer dem gleichen Beispiel erläutert. Dabei soll es darum gehen, von unterschiedlichen Programmen aus Daten in eine Testdatei zu schreiben, deren numerischer Primärschlüssel lückenlos aufsteigend vergeben werden soll. In einer zweiten Datei mit dem Namen `Counter` findet sich der jeweils aktuelle Wert für den Schlüssel der Testdatei. Um also einen Satz zu erzeugen, liest die Anwendung die `Counter`-Datei und legt mit dem dort gefundenen Wert einen neuen Satz in der Testdatei an. Daraufhin erhöht sie den Wert um 1 und schreibt ihn in die `Counter`-Datei zurück. Lesen nun zwei konkurrierende Anwendungen denselben Datensatz zur selben Zeit, werden sie ebenso gleichzeitig versuchen, einen Satz in der Testdatei mit demselben Primärschlüssel anzulegen, was in mindestens einem Fall zu einer Schlüsselverletzung führt. Am Ende soll jede Anwendung ausgeben, wie oft es zu diesem Fehler kam. Schon jetzt ist klar, dass dieser Wert für eine professionelle Anwendung immer null sein muss.

## File Connector

Ein *File Connector* ist ein interner Speicherbereich, der einem Dateinamen zugeordnet und für den Anwendungsentwickler nicht zugänglich ist. Er wird von der COBOL-Laufzeitumgebung für die aktuelle Anwendung erzeugt und verwaltet unterschiedliche Informationen über die zugehörige Datei. Dazu gehören neben dem aktuellen Dateistatus auch die aktuelle Dateiposition oder beispielsweise der `OPEN`-Modus. Vor allem aber, und das ist für dieses Kapitel von besonderem Interesse, verwaltet der File Connector auch die Liste aller Datensätze, die er in der Datei gesperrt hat. Jede gestartete Anwendung, die eine Datei öffnet, hat einen eigenen File Connector pro Datei. Dadurch ist es möglich, dass mehrere Anwendungen gleichzeitig unterschiedliche Satzsperren auf ein und dieselbe Datei halten können,

selbst wenn sie im selben Hauptspeicher ausgeführt werden und es sich letztendlich immer um dasselbe COBOL-Programm handelt, das mehrfach gestartet wurde.

## Verarbeitung ohne Satzsperren

Um sicherzustellen, dass immer nur ein User gleichzeitig die Daten eines Kunden ändern kann, muss die Anwendung wissen, ob der gewünschte Datensatz derzeit von einem anderen Benutzer bearbeitet wird oder nicht. Damit wird die Programmierung aufwendiger, da man sich Gedanken für den Fall machen muss, dass man aktuell nicht alleine zugreift. Dieser Abschnitt soll aufzeigen, was passiert, wenn man sich um den konkurrierenden Zugriff keine Gedanken macht.

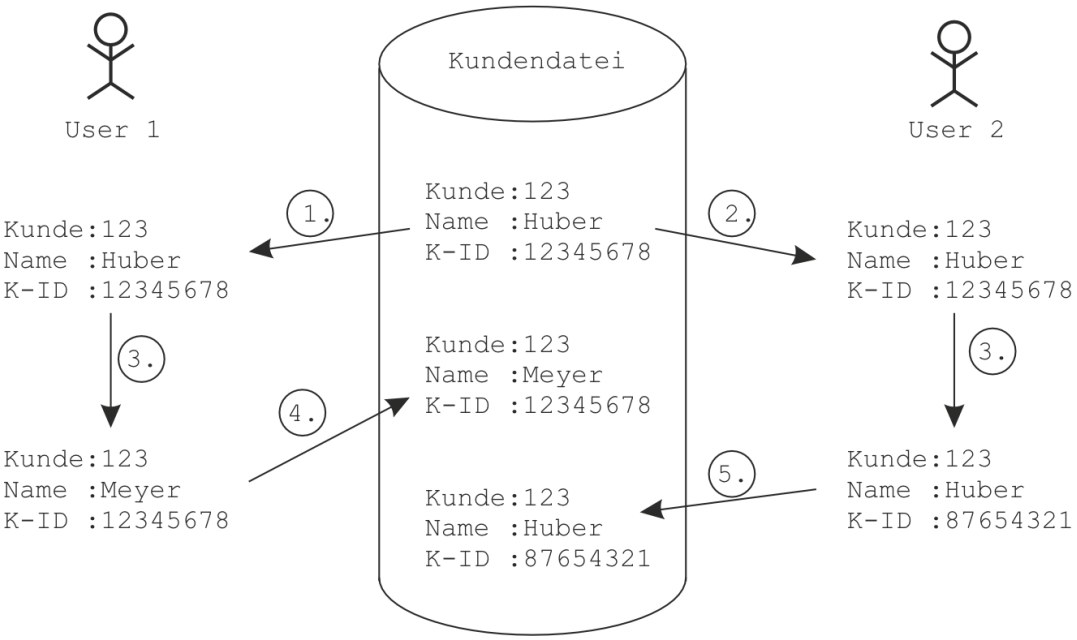


*Abbildung 1: Gemeinsamer Zugriff ohne Satzsperren*

In Abbildung 1 sind zwei User dargestellt, die gleichzeitig den Kunden mit der Nummer 123 bearbeiten wollen.

1. User 1 liest den Satz und bekommt ihn in seine Anwendung übertragen. Der Kunde heißt Huber und hat die Kunden-ID 12345678.
2. User 2 greift ebenfalls auf diesen Satz zu und bekommt genau dieselben Daten.
3. Beide Anwender ändern die ihnen zur Verfügung gestellten Informationen. User 1 verändert den Namen, der Kunde heißt jetzt Meyer. User 2 nimmt Einfluss auf die Kunden-ID und ändert sie auf 87654321.
4. User 1 schreibt den Datensatz zurück, was auch problemlos funktioniert. Er ist der Überzeugung, seine Änderung erfolgreich durchgebracht zu haben. In der Datei steht tatsächlich der Kunde Meyer mit der Kunden-ID 12345678.
5. Danach schreibt auch User 2 seine Daten. Auch er bekommt die Information, dass alles ordnungsgemäß funktioniert hat. Das Problem ist nur, dass der

Kunde in der Datei jetzt wieder Huber heißt, weil User 2 nichts von der Änderung von User 1 mitbekommen hat.

In einer professionellen Anwendung darf es niemals zu der eben beschriebenen Situation kommen.

### Beispiel: Counter

Im Folgenden soll aufgezeigt werden, was passiert, wenn man das aus dem Anfang des Kapitels bekannte Problem mit der `Counter`-Datei ohne Satzsperren realisiert. Werden weder bei der Dateibeschreibung in der `SELECT`-Klausel noch bei irgendeiner Anweisung für den Dateizugriff irgendwelche Angaben für den Fall des konkurrierenden Zugriffs gemacht, bedeutet dies, dass eine Anwendung eine Datei exklusiv öffnen will. Für das vorliegende Problem würde dies bedeuten, dass eine zweite konkurrierende Anwendung erst gar nicht gestartet werden kann, beziehungsweise diese eine Fehlermeldung erhält, wenn sie versucht, die Datei zu öffnen. Das bedeutet, dass ein User alle Datensätze aller Dateien, mit denen er arbeitet, gleichzeitig sperrt.

## SHARING-Angabe

Um ein gemeinsames Arbeiten überhaupt zu ermöglichen, ist es notwendig, bei der `OPEN`-Anweisung eine entsprechende `SHARING`-Angabe zu machen.

```
OPEN { { INPUT  } [sharing-phrase] [ retry-phrase]
       { OUTPUT }
       { I-O    }
       { EXTEND }

                { Dateiname-1 [WITH NO REWIND] }... } ...

 sharing-phrase:
                 {ALL OTHER }
   SHARING WITH  {NO OTHER  }
                 {READ ONLY }

 retry-phrase:
          {arithmetischer-Ausdruck-1  TIMES  }
   RETRY  {FOR arithmetischer-Ausdruck-2 SECONDS }
          {FOREVER                            }
```

*Abbildung 2: `OPEN`-Anweisung*

## SHARING WITH ALL OTHER

Öffnet eine Anwendung eine Datei mit diesem Zusatz, erlaubt sie es anderen Anwendungen, sie gleichzeitig zu öffnen. Versucht die konkurrierende Anwendung, die Datei mit `OPEN OUTPUT` anzulegen, bekommt sie eine Fehlermeldung.

## SHARING WITH NO OTHER

Hier öffnet eine Anwendung die Datei exklusiv. Jeder Versuch einer konkurrierenden Anwendung, die Datei ebenfalls zu öffnen, wird als fehlerhaft zurückgewiesen.

## SHARING WITH READ ONLY

Damit erlaubt die öffnende Anwendung ihren Konkurrenten, gleichzeitig lediglich lesend zuzugreifen. Nur wenn diese einen `OPEN INPUT` versuchen, können sie Erfolg haben.

In Abbildung 3 sind alle erlaubten beziehungsweise nicht erlaubten Möglichkeiten für einen konkurrierenden Zugriff beschrieben. Sollte eine `OPEN`-Anweisung aufgrund eines Konfliktes mit einer anderen Anwendung nicht ausgeführt werden, kann mithilfe der `RETRY`-Angabe bestimmt werden, ob und wie viele weitere Versuche unternommen werden sollen. Auf die Syntax der `RETRY`-Angabe wird im Folgenden noch genauer eingegangen.

| OPEN Anweisung im aktuellen Programm | | Bereits von einer anderen Anwendung durchgeführte OPEN Anweisung | | | | |
|---|---|---|---|---|---|---|
| | | SHARING WITH NO OTHER | SHARING WITH READ ONLY | | SHARING WITH ALL OTHER | |
| | | extend i-o input output | extend i-o output | input | extend i-o output | input |
| SHARING WITH NO OTHER | extend i-o input output | nicht erlaubt | nicht erlaubt | nicht erlaubt | nicht erlaubt | nicht erlaubt |
| SHARING WITH READ ONLY | extend i-o | nicht erlaubt | nicht erlaubt | nicht erlaubt | nicht erlaubt | erlaubt |
| | input | nicht erlaubt | nicht erlaubt | erlaubt | nicht erlaubt | erlaubt |
| | output | nicht erlaubt | nicht erlaubt | nicht erlaubt | nicht erlaubt | nicht erlaubt |
| SHARING WITH ALL OTHER | extend i-o | nicht erlaubt | nicht erlaubt | nicht erlaubt | erlaubt | erlaubt |
| | input | nicht erlaubt | erlaubt | erlaubt | erlaubt | erlaubt |
| | output | nicht erlaubt | nicht erlaubt | nicht erlaubt | nicht erlaubt | nicht erlaubt |

*Abbildung 3: Erlaubte konkurrierende OPEN-Anweisungen*

**Hinweis**

Soll eine Datei mit einer SHARING-Angabe geöffnet werden, ist es zusätzlich notwendig, in ihrer SELECT-Klausel eine Angabe darüber zu treffen, wie die einzelnen Datensätze gesperrt werden sollen.

## LOCK-MODE-Zusatz

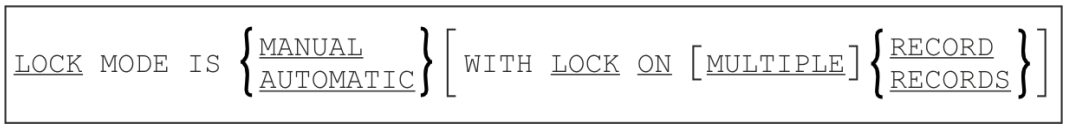


*Abbildung 4: LOCK MODE-Zusatz der SELECT-Klausel*

### LOCK MODE IS MANUAL

Bei dieser Angabe wird nur dann ein Datensatz gesperrt, wenn dies explizit in der entsprechenden IO-Anweisung verlangt wird. Ein Beispiel dafür wäre ein `READ WITH LOCK`.

### LOCK MODE IS AUTOMATIC

Ohne dass sich der Anwendungsentwickler weiter darum kümmert, würde bei dieser Angabe ein Datensatz in der Datei alleine dadurch gesperrt werden, dass er gelesen wird. Eine normale `READ`-Anweisung reicht aus, um eine Satzsperre anzubringen. Dies kann bei fortlaufender Verarbeitung dazu führen, dass immer mehr Sätze von einem User gesperrt werden und ein konkurrierender Zugriff immer weniger möglich wird. In einem Testumfeld mag dies noch nicht besonders ins Gewicht fallen. Sobald das Programm dann aber stundenlang in der Praxis eingesetzt wird, tritt dieser unangenehme Effekt auf.

### LOCK ON RECORDS

Diese Angabe gilt standardmäßig und sorgt dafür, dass eine Anwendung gleichzeitig immer nur einen Datensatz der zugehörigen Datei sperren kann. Jede folgende Dateioperation (außer `START`) hebt die Satzsperre auf. Damit kann das gefährliche Verhalten von `LOCK MODE AUTOMATIC` zwar entschärft werden, es ist jedoch genau zu prüfen, ob es für die Programmlogik genügt, immer nur einen Satz gleichzeitig im Zugriff zu haben.

### LOCK ON MULTIPLE RECORDS

Damit wird es möglich, dass eine Anwendung gleichzeitig mehrere Datensätze einer Datei sperrt. Diese können dann entweder einzeln durch ein `REWRITE WITH NO LOCK` oder alle mithilfe der `UNLOCK`-Anweisung wieder freigegeben werden.

## Beispiel: Counter ohne Satzsperren

Das Beispiel in Listing 1 zeigt den Versuch, das `Counter`-Problem ohne Satzsperren zu lösen.

```
 1 identification division.
 2 program-id. OhneSperre.
 3 environment division.
 4 input-output section.
 5 file-control.
 6     select counter assign to "counter1"
 7         organization is indexed
 8         access dynamic
 9         record key schluessel
10         lock mode is manual
```

```
11          file status is count-stat.
12      select testdatei assign to "test1"
13          organization is indexed
14          access dynamic
15          record key testkey
16          lock mode is manual
17          file status is test-stat.
18  data division.
19  file section.
20  fd  counter.
21  01  counter-satz.
22      05  schluessel             pic 999.
23      05  wert                   pic 9(5).
24  fd  testdatei.
25  01  testsatz.
26      05  testkey                pic 9(5).
27      05  testdaten              pic x(20).
28  working-storage section.
29  01  count-stat                 pic xx.
30      88  count-ok               value "00" thru "09".
31  01  test-stat                  pic xx.
32      88  test-ok                value "00" thru "09".
33  01  anzahlFehlversuche         pic 9(5) value 0.
34  procedure division.
35  counter-auslesen.
36      open i-o
37           sharing with all other
38           counter testdatei
39      if count-ok and test-ok
40          move 100 to schluessel
41          perform 10000 times
42              read counter
43                  invalid key
44                      perform counter-satz-erzeugen
45              end-read
46              if count-ok
47                  perform testsatz-erzeugen
48              else
49                  display "Read counter:" count-stat
50              end-if
51          end-perform
52          close counter testdatei
53      else
54          display "Open counter:" count-stat
55                  " Open test: " test-stat
56      end-if
57      display "Anzahl Fehlversuche:" anzahlFehlversuche
58      stop run.
```

```
59 counter-satz-erzeugen.
60     *> Wenn die Datei noch nicht vorhanden war, wird
61     *> hier der Datensatz mit der Nummer 100 und dem
62     *> Anfangswert 1 erzeugt.
63     move 1 to wert
64     write counter-satz
65     if not count-ok
66         display "Write-counter:"
67             count-stat
68     end-if
69     .
70 testsatz-erzeugen.
71     move wert to testkey
72     move "Testdaten" to testdaten
73     write testsatz
74     if test-ok
75         display "ok:" testkey
76         add 1 to wert
77         rewrite counter-satz
78     else
79         display "Write-testsatz:"
80             test-stat
81         add 1 to anzahlFehlversuche
82     end-if
83     .
```

*Listing 1: Konkurrierender Zugriff ohne Satzsperren*

Sobald man es gleichzeitig mehrfach startet, wird deutlich, dass immer nur ein User in der Lage ist, einen Testdatensatz zu schreiben. Obwohl pro Programm 10.000 Sätze geschrieben werden sollen, werden es am Ende weit weniger sein, weil jeder Fehlversuch abgezogen werden muss.

## Pessimistische Satzsperren

Um das Problem des konkurrierenden Zugriffs vernünftig zu lösen, ist es unabdingbar, Datensätze zu sperren, um so sicherzustellen, dass immer nur ein Anwender gleichzeitig einen Datensatz bearbeitet.

Diese Satzsperren können entweder pessimistisch oder optimistisch angebracht werden. Von pessimistischer Satzsperre (oder pessimistischem Locking) spricht man, wenn ein Datensatz schon beim ersten Lesen für einen User gesperrt wird. Erst wenn er ihn geändert zurückschreibt oder explizit angibt, dass er ihn nicht weiter bearbeiten will, wird der Satz wieder freigegeben. Der Vorteil dieser Vorgehensweise liegt darin, dass die Programmlogik relativ einfach bleibt. Der Hauptnachteil ist jedoch, dass ein Satz so lange nicht bearbeitet werden kann, wie er bei einem anderen Anwender auf dem Bildschirm steht. Dabei muss durchaus berücksichtigt

werden, dass dieser in die Mittagspause geht oder sich sein Rechner »aufgehängt« hat.

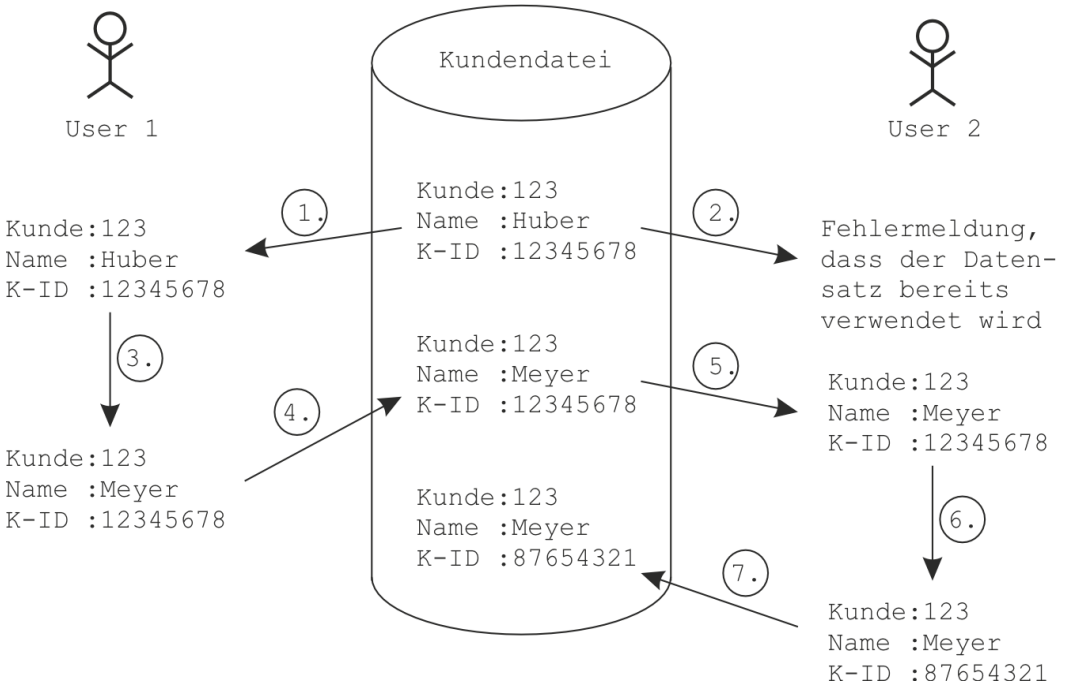


*Abbildung 5: Zugriff bei pessimistischem Locking*

In Abbildung 5 ist dargestellt, was bei pessimistischen Satzsperren passiert, wenn zwei Benutzer gleichzeitig den Kundensatz 123 bearbeiten wollen.

1. User 1 liest den Kunden mit der Nummer 123. Er heißt Huber und hat die Kunden-ID 12345678.
2. User 2 versucht ebenfalls, diesen Kundensatz zu lesen. Nun kommt es ganz darauf an, wie die Anwendung programmiert ist. Bei einer Online-Anwendung macht es keinen Sinn, so lange zu warten, bis der Satz wieder frei ist. Der Benutzer müsste eventuell minutenlang (oder noch länger) warten, bis seine Anwendung auf die Anforderung des Kundensatzes reagiert. Er wird den Eindruck bekommen, dass seine Anwendung nicht mehr richtig funktioniert und unter Umständen das Programm oder den ganzen Computer neu starten. Daher scheint es besser zu sein, ihm sofort mitzuteilen, dass der Datensatz gerade nicht zur Verfügung steht.
3. User 1 kann die Daten in Ruhe ändern.
4. Erst wenn der Datensatz ordentlich in die Datei zurückgeschrieben wurde, ist er freigegeben.
5. Nun kann auch User 2 zugreifen. Eventuell hat er so lange gewartet oder er versucht es nach einiger Zeit erneut.
6. Durch dieses Verfahren ist in jedem Fall sichergestellt, dass User 2 die aktuellste Version des Datensatzes bekommt, die die Änderungen von User 1 beinhaltet.

Während er nun die Daten seinerseits manipuliert, sperrt er den Satz für alle anderen.

7. Schließlich schreibt auch User 2 den Satz zurück, an dem sich insgesamt sowohl der Name als auch die Kunden-ID geändert haben.

Um ein solches Verhalten programmieren zu können, ist es notwendig, die Datei in der `SELECT`-Klausel mit einer `LOCK MODE`-Angabe zu versehen. Außerdem muss die Datei bei der `OPEN`-Anweisung mit einer geeigneten `SHARING`-Angabe geöffnet werden. Die entsprechenden Syntaxbeschreibungen befinden sich im Abschnitt »Verarbeitung ohne Satzsperren«.

## Manuelle Einzelsatzsperre

Wurde die Datei mit `LOCK MODE MANUAL` und dem optionalen Zusatz `LOCK ON RECORDS` versehen, wird erst dann ein Satz gesperrt, wenn dies explizit in der entsprechenden Anweisung verlangt wird. Relevant ist das bei den Anweisungen `READ`, `WRITE` und `REWRITE`, deren Syntax hinsichtlich des Sperrens von Datensätzen im Folgenden beleuchtet wird. Außerdem kann immer nur ein Satz pro File Connector gesperrt werden. Eine erfolgreiche Beendigung einer beliebigen IO-Anweisung (außer `START`) hebt eine eventuell vorhandene Sperre dieses File Connectors auf einen beliebigen Satz der zugehörigen Datei auf.

```
READ Dateiname-1 { NEXT     } RECORD [ INTO Bezeichner-1]
                 { PREVIOUS }

 [ ADVANCING ON LOCK ]
 [ IGNORING LOCK     ]
 [ retry-phrase      ]

 [ WITH LOCK    ]
 [ WITH NO LOCK ]

 [ | AT END unbedingte-Anweisung-1     | ]
 [ | NOT AT END unbedingte-Anweisung-2 | ]

 [ END-READ ]

 retry-phrase:
          { arithmetischer-Ausdruck-1  TIMES   }
   RETRY  { FOR arithmetischer-Ausdruck-2 SECONDS }
          { FOREVER                             }
```

*Abbildung 6: `READ`-Anweisung zum sequenziellen Lesen*

In `READ NEXT` oder `READ PREVIOUS` können verschiedene Angaben im Zusammenhang mit Datensatzsperren vorgenommen werden, die im Folgenden beschrieben werden.

## ADVANCING ON LOCK

Damit wird definiert, dass der eigentlich nächste Datensatz übersprungen werden soll, wenn er von einer anderen Anwendung gesperrt ist. Dies wird so lange angewendet, bis ein freier Satz oder das Dateiende gefunden wird. Damit ist es möglich, dass mehrere Programme parallel eine Anforderungsdatei abarbeiten, wobei jedes einzelne den Satz sperrt, den es gerade gelesen hat.

## IGNORING LOCK

Ein `READ IGNORING LOCK` liest den gewünschten Satz auch dann, wenn er gleichzeitig von jemand anderem gesperrt wurde. Damit kann man einen Datensatz auch dann lesen, wenn er gerade in Bearbeitung ist. Ein gleichzeitiges Sperren mit der Angabe `WITH LOCK` ist jedoch in keinem Fall möglich.

## RETRY

Sollte der gewünschte Datensatz oder die gesamte Datei gesperrt sein, kann mit diesem Zusatz festgelegt werden, wie zu verfahren ist.

### Anzahl TIMES

Nachdem der Zugriff gescheitert ist, sollen so viele weitere Versuche unternommen werden, wie der arithmetische Ausdruck angibt. Wie viel Zeit zwischen jedem Versuch liegen soll, wird vom Compilerhersteller bestimmt. Sind alle Versuche gescheitert, wird die zugehörige IO-Anweisung beendet und der Dateistatus auf den Wert 51 gesetzt.

### FOR Anzahl SECONDS

Nachdem der erste Versuch gescheitert ist, soll die hier angegebene Zeit lang weiter versucht werden, auf den Satz zuzugreifen. Wie viele Zugriffe pro Sekunde ausgeführt werden, legt der Compilerhersteller fest. Sind alle Versuche gescheitert, wird die zugehörige IO-Anweisung beendet und der Dateistatus auf den Wert 51 gesetzt.

### FOREVER

Es soll so lange auf den Datensatz gewartet werden, bis er verfügbar ist.

## WITH LOCK

Bestimmt ausdrücklich, dass der gewünschte Satz gesperrt werden soll. Bei manueller Einzelsatzsperre hebt dieser Zusatz eine eventuell vorhandene Satzsperre für dieselbe Datei auf, sodass pro File Connector nur maximal ein Satz gesperrt bleibt.

## WITH NO LOCK

Der Satz soll zwar gelesen, aber nicht gesperrt werden. Bei manueller Einzelsatzsperre hebt dieser Zusatz eine eventuell vorhandene Satzsperre für dieselbe Datei auf, sodass jetzt kein einziger Satz mehr von dem entsprechenden File Connector gesperrt ist. Fehlt die `LOCK`-Angabe, wird bei manueller Satzsperre immer `WITH NO LOCK` postuliert.

## READ-Anweisung zum wahlfreien Lesen

```
READ Dateiname-1 RECORD [INTO Bezeichner-1]

 [IGNORING LOCK  ]
 [retry-phrase   ]

 [WITH LOCK    ]
 [WITH NO LOCK ]

 [KEY IS {Datenname-1  }]
         {Record-Key-1 }

 [[INVALID KEY unbedingte-Anweisung-1    ]]
 [[NOT INVALID KEY unbedingte-Anweisung-2]]

 [END-READ]

retry-phrase:

       {arithmetischer-Ausdruck-1  TIMES  }
 RETRY {FOR arithmetischer-Ausdruck-2 SECONDS}
       {FOREVER                             }
```

*Abbildung 7: `READ`-Anweisung zum wahlfreien Lesen*

Auch bei der `READ`-Anweisung zum wahlfreien Lesen finden sich die Angaben `IGNORING LOCK`, `RETRY` und `WITH LOCK` beziehungsweise `WITH NO LOCK`. Sie haben dieselbe Bedeutung wie bei einem sequenziellen `READ`.

## WRITE- und REWRITE-Anweisung

```
WRITE { Satzname      } [ FROM { Bezeichner-1 } ]
      { FILE Dateiname }       { Literal-1   }

  [ retry-phrase ]

  [ WITH LOCK    ]
  [ WITH NO LOCK ]

  [ | INVALID KEY unbedingte-Anweisung-1     | ]
  [ | NOT INVALID KEY unbedingte-Anweisung-2 | ]

  [ END-WRITE ]

 retry-phrase:

        { arithmetischer-Ausdruck-1  TIMES   }
  RETRY { FOR arithmetischer-Ausdruck-2 SECONDS }
        { FOREVER                              }
```

*Abbildung 8: `WRITE`-Anweisung*

Bei der `WRITE`-Anweisung regelt die `RETRY`-Angabe, wie verfahren werden soll, wenn der zu schreibende Satz von einer anderen Anwendung gesperrt ist. Wurde die Sperre von dem eigenen File Connector angebracht, kann in jedem Fall geschrieben werden.

Wird der Zusatz `WITH LOCK` definiert, wird der Datensatz gesperrt, wenn er erfolgreich geschrieben werden konnte. Im Fall der manuellen Einzelsperre wird ein eventuell bereits von diesem File Connector gesperrter Satz in derselben Datei aufgehoben.

`WITH NO LOCK` schreibt den Satz ohne weitere Sperre, was dem Standardverhalten entspricht, wenn bei manueller Einzelsperre keinerlei `LOCK`-Angabe gemacht wurde. Dann liegt auch keine weitere Satzsperre für den zugehörigen File Connector vor.

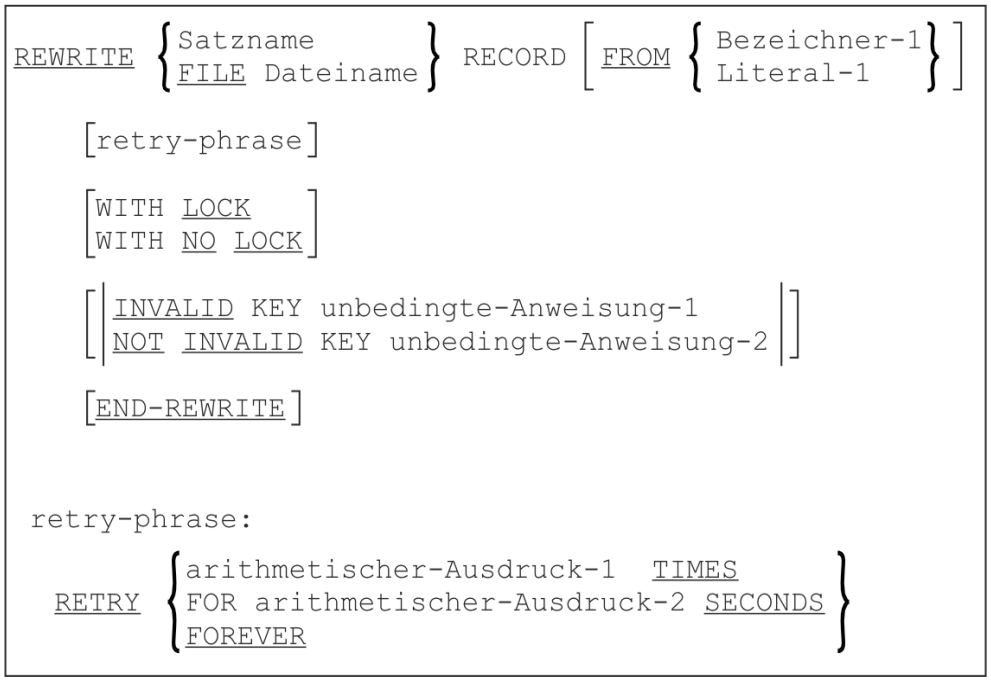


*Abbildung 9: `REWRITE`-Anweisung*

Hinsichtlich des Verhaltens bei gesperrten Sätzen und des Veranlassens eigener Sperren unterscheidet sich die `REWRITE`-Anweisung nicht von der `WRITE`-Anweisung.

**Hinweis**

Spätestens durch die `CLOSE`-Anweisung oder bei Programmende werden alle Satzsperren der betreffenden Anwendung wieder freigegeben.

## Manuelle Mehrfachsatzsperre

Wurde eine Datei in der `SELECT`-Klausel mit dem Zusatz `LOCK MODE IS MANUAL WITH LOCK ON MULTIPLE RECORDS` angegeben, definiert das manuelle Mehrfachsatzsperren für diese Datei. Um einen Satz zu sperren, muss dies wie bei der manuellen Einzelsatzsperre durch die explizite Angabe von `WITH LOCK` bei der entsprechenden IO-Anweisung verlangt werden.

Der wesentliche Unterschied zur Einzelsatzsperre liegt darin, dass pro File Connector beliebig viele Sätze in einer Datei gleichzeitig gesperrt sein können. Jede IO-Anweisung, die den Zusatz `WITH LOCK` trägt, erweitert die Menge der gesperrten Sätze um eins. Wurde keine `LOCK`-Angabe gemacht oder ist explizit `WITH NO LOCK` angegeben und wurde der Satz von der aktuellen Anwendung gesperrt, wird nur dieser eine Datensatz wieder freigegeben.

Arbeitet eine Anwendung mit Mehrfachsatzsperren, muss sie sehr genau aufpassen, wann sie welche Sätze sperrt und vor allem, wann sie sie wieder freigibt. Um sicherzugehen, dass für eine Datei keinerlei Sperren mehr existieren, kann die UNLOCK-Anweisung verwendet werden.


*Abbildung 10: UNLOCK-Anweisung*

Auch hier gilt, dass spätestens durch die CLOSE-Anweisung oder durch das Programmende alle Sperren der betreffenden Anwendung wieder gelöscht werden.

## Automatische Einzelsatzsperre

Soll eine Datei mithilfe der automatischen Einzelsatzsperre verarbeitet werden, muss sie in ihrer SELECT-Klausel den Zusatz LOCK MODE IS AUTOMATIC und optional LOCK ON RECORDS tragen. In diesem Fall gilt, dass keine IO-Anweisung, die auf diese Datei ausgeführt wird, einen der Zusätze IGNORING LOCK, WITH LOCK oder WITH NO LOCK tragen darf. Jede erfolgreiche READ-Anweisung sperrt den soeben gelesenen Satz, und da es sich um eine Einzelsatzsperre handelt, gibt sie einen eventuell vorher durch den File Connector gesperrten Satz wieder frei. Durch Anweisungen wie WRITE oder REWRITE werden keine weiteren Sätze gesperrt, sondern vielmehr eine eventuell vorhandene Sperre eines beliebigen Satzes der entsprechenden Datei freigegeben. Programmiert man also ein READ und dann ein REWRITE für denselben Satz, wird dieser zunächst durch das READ gesperrt und dann mittels REWRITE zurückgeschrieben und freigegeben.

UNLOCK hebt, wie CLOSE, alle Sperren in einer Datei auf, die durch den zugehörigen File Connector erzeugt wurden. Spätestens durch das Programmende ist sichergestellt, dass die entsprechende Anwendung keinen Satz mehr gesperrt hat.

## Automatische Mehrfachsatzsperre

Eine automatische Mehrfachsperre definiert man durch den Zusatz LOCK MODE IS AUTOMATIC WITH LOCK ON MULTIPLE RECORDS in der SELECT-Klausel.

Auch hier gilt, dass keine IO-Anweisung, die auf die Datei ausgeführt wird, einen der Zusätze IGNORING LOCK, WITH LOCK oder WITH NO LOCK tragen darf, und dass ein Satz immer dann gesperrt wird, wenn eine erfolgreiche READ-Anweisung durchgeführt werden konnte. Eine automatische Freigabe gesperrter Sätze durch ein weiteres READ oder eine beliebige andere IO-Anweisung findet jedoch nicht statt.

Erst durch die UNLOCK- oder CLOSE-Anweisung werden Satzsperren aufgehoben. Bei Programmende sind alle Sperren der entsprechenden Anwendung obsolet.

## Beispiel: Counter mit pessimistischen Satzsperren

In folgendem Beispiel soll das Counter-Problem, das am Anfang des Kapitels beschrieben wurde, mithilfe pessimistischer Satzsperren gelöst werden. Da es sich um eine reine Batch-Anwendung handelt, macht dies durchaus Sinn. Sobald jedoch ein Benutzer im Spiel ist, sollte überlegt werden, ob optimistische Satzsperren nicht besser wären.

Das Beispiel arbeitet mit manuellen Einzelsatzsperren. Außerdem wird durch die OPEN-Anweisung bestimmt, dass bei einem Zugriff stets so lange gewartet werden soll, bis der entsprechende Datensatz wieder frei ist. Erreicht wird dies durch den Zusatz RETRY FOREVER bei der OPEN-Anweisung.

```
 1 identification division.
 2 program-id. Pessimistisch.
 3 environment division.
 4 input-output section.
 5 file-control.
 6     select counter assign to "counter2"
 7         organization is indexed
 8         access dynamic
 9         record key schluessel
10         lock mode is manual
11         file status is count-stat.
12     select testdatei assign to "test2"
13         organization is indexed
14         access dynamic
15         record key testkey
16         lock mode is manual
17         file status is test-stat.
18 data division.
19 file section.
20 fd  counter.
21 01  counter-satz.
22     05  schluessel             pic 999.
23     05  wert                   pic 9(5).
24 fd  testdatei.
25 01  testsatz.
26     05  testkey                pic 9(5).
27     05  testdaten              pic x(20).
28 working-storage section.
29 01  count-stat                 pic xx.
30     88  count-ok               value "00" thru "09".
31 01  test-stat                  pic xx.
```

```
32     88  test-ok                 value "00" thru "09".
33 01  anzahlFehlversuche          pic 9(5) value 0.
34 procedure division.
35 counter-auslesen.
36     open i-o
37          sharing with all other
38          retry forever
39          counter testdatei
40     if count-ok and test-ok
41         move 100 to schluessel
42         perform 10000 times
43             read counter with lock
44                 invalid key
45                     perform counter-satz-erzeugen
46             end-read
47             if count-ok
48                 perform testsatz-erzeugen
49             else
50                 display "Read counter:" count-stat
51             end-if
52         end-perform
53         close counter testdatei
54     else
55         display "Open counter:" count-stat
56                 " Open test: " test-stat
57     end-if
58     display "Anzahl Fehlversuche:" anzahlFehlversuche
59     stop run.
60 counter-satz-erzeugen.
61     *> Wenn die Datei noch nicht vorhanden war, wird
62     *> hier der Datensatz mit der Nummer 100 und dem
63     *> Anfangswert 1 erzeugt.
64     move 1 to wert
65     write counter-satz
66     if not count-ok
67         display "Write-counter:"
68             count-stat
69     end-if
70     .
71 testsatz-erzeugen.
72     move wert to testkey
73     move "Testdaten" to testdaten
74     write testsatz
75     if test-ok
76         display "ok:" testkey
77         add 1 to wert
78         rewrite counter-satz
79     else
```

```
80          display "Write-testsatz:"
81              test-stat
82          add 1 to anzahlFehlversuche
83      end-if
84      .
```

*Listing 2: Konkurrierender Zugriff mit pessimistischen Satzsperren*

Bei der Ausführung des vorliegenden Beispiels darf es am Ende zu keinem einzigen Fehlversuch gekommen sein. Außerdem müssen sich exakt 10.000 Sätze pro gestarteter Anwendung in der Testdatei befinden. Lässt man das Beispiel also zweifach parallel laufen, befinden sich am Ende in der Testdatei 20.000 Datensätze.

## Optimistische Satzsperren

Wie aus den vorangegangenen Ausführungen deutlich geworden ist, stellt eine Online-Anwendung besondere Ansprüche an das Sperren von Datensätzen. Es ist dem Anwender nicht zumutbar, dass er eine unbestimmte Zeit lang warten muss, bis seine Anwendung ihm den gewünschten Datensatz präsentiert. Aber auch die sofortige Information, dass der Satz gerade von einem Kollegen »bearbeitet« wird, ist nicht immer befriedigend. Derart programmierte Systeme unterscheiden häufig, ob Informationen nur angezeigt oder eventuell auch verändert werden sollen. Bei Ersterem wird man mit einem `READ IGNORING LOCK` arbeiten und von der Programmlogik her dafür sorgen, dass keine Änderungen zurückgeschrieben werden können. Stellt der Anwender bei der Betrachtung der Daten jedoch fest, dass er sie ändern muss, ist er gezwungen, die aktuelle Verarbeitung abzubrechen, um den Satz erneut, diesmal jedoch für die Bearbeitung, zu laden. Passiert dies pro Tag öfter, wird er irgendwann dazu übergehen, den Satz sofort im Bearbeitungsmodus anzusehen, weil er dann meist auch ohne Änderungen durchzuführen abbrechen kann.

Optimistische Satzsperren gehen davon aus, dass permanent Daten abgefragt werden, es jedoch nur sehr selten vorkommt, dass zwei Benutzer zeitgleich denselben Satz ändern wollen. Daher werden die Informationen zunächst ohne jegliche Sperren gelesen und dem User angezeigt. Nur wenn er tatsächlich eine Änderung durchgeführt hat und diese speichern will, wird der Datensatz vom Programm gelesen und gesperrt und unmittelbar danach mit den Änderungen wieder zurückgeschrieben und dadurch freigegeben. Der Satz war für weniger als eine Sekunde gesperrt.

Was ist aber, wenn doch ein anderer User den Satz zwischenzeitlich geändert hat? Um dies festzustellen, fügt man in die Satzstruktur häufig eine Versionsnummer oder einen Timestamp ein. Soll dann eine Änderung zurückgeschrieben werden, kann man aufgrund dieses Kennzeichens feststellen, dass die eigene Datenbasis nicht mehr aktuell ist, und dies dem Anwender mitteilen. Er kann nun die Daten

neu lesen und seine Änderungen auf dem aktuellen Stand durchführen. Passiert ihm das pro Tag öfter, wird er aber auch unzufrieden sein. Die Art des Sperrverhaltens ist also sehr stark von der konkreten Anwendung abhängig.

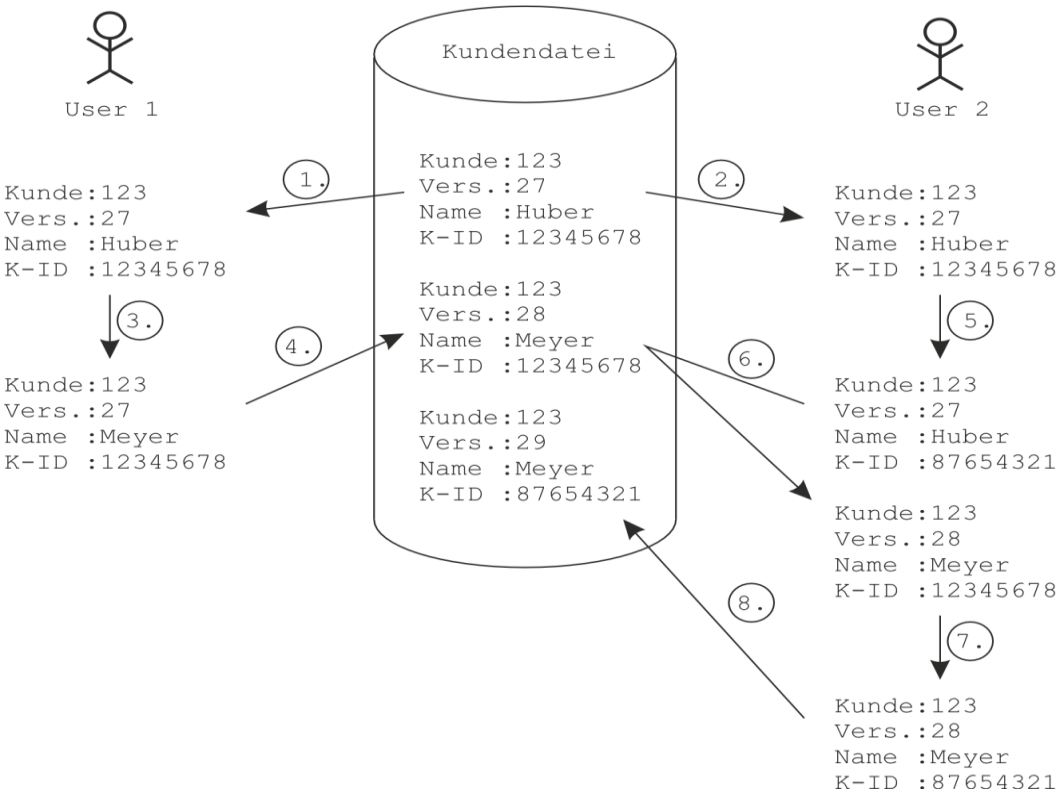


*Abbildung 11: Zugriff bei optimistischem Locking*

Die Vorgänge bei optimistischen Satzsperren lassen sich wie folgt erklären:

1. User 1 liest den Kunden mit der Nummer 123, ohne den Satz zu sperren.
2. Auch User 2 bekommt den gewünschten Satz, der von ihm nicht gesperrt wird.
3. User 1 ändert den Namen Huber auf Meyer.
4. Sobald der geänderte Datensatz gespeichert werden soll, liest ihn das COBOL-Programm und sperrt ihn. Daraufhin wird die Versionsnummer verglichen. Da es sich immer noch um Version 27 handelt, werden die Änderungen durchgeführt und die Versionsnummer erhöht. Der ganze Vorgang dauert wenige Millisekunden. Der Satz ist jetzt nicht mehr gesperrt.
5. User 2 ändert unterdessen die Kunden-ID.
6. Nun versucht User 2, seine Änderungen durchzubekommen. Auch seine Anwendung liest den Datensatz und sperrt ihn. Da sich die Versionsnummer aber geändert hat, informiert er den Benutzer und gibt den Satz, ohne ihn geändert zu haben, wieder frei.
7. User 2 kann erneut die Kunden-ID ändern. Es wäre jedoch besser, wenn sich die Anwendung seine Änderungen gemerkt hätte, um sie automatisch auf den

neu gelesenen Satz mit der Version 28 anzuwenden. Der Benutzer bräuchte die Daten lediglich noch zu kontrollieren, bevor er der Änderung zustimmt.

8. Erneut versucht User 2, seine Änderungen zu speichern. Es läuft das bereits bekannte Prozedere ab: Der Satz wird gelesen und gesperrt. Da die Versionsnummer übereinstimmt, werden die Änderungen durchgeführt und die Version erhöht. Der Satz wird als Ganzes zurückgeschrieben und freigegeben.

## Beispiel: Counter mit optimistischen Satzsperren

Abermals soll das `Counter`-Beispiel vom Anfang des Kapitels programmiert werden. Da es diesmal mithilfe optimistischer Satzsperren gelöst wird, wurde der Datensatz der `Counter`-Datei um das Feld `Version` erweitert. Es ist fünfstellig numerisch, was jederzeit ausreicht, weil für die Logik nicht die tatsächliche Version wichtig ist, sondern lediglich die Tatsache, dass sich die Versionsnummer verändert hat. Irgendwann wird das Feld überlaufen und die Zählung beginnt wieder mit null.

Um festzustellen, dass sich die Versionsnummer seit dem ersten Lesen geändert hat, werden die Daten in einen Bereich der `WORKING-STORAGE SECTION` kopiert. Dies kann elegant über einen `READ INTO` erfolgen.

```
 1 identification division.
 2 program-id. Pessimistisch.
 3 environment division.
 4 input-output section.
 5 file-control.
 6     select counter assign to "counter3"
 7         organization is indexed
 8         access dynamic
 9         record key schluessel
10         lock mode is manual
11         file status is count-stat.
12     select testdatei assign to "test3"
13         organization is indexed
14         access dynamic
15         record key testkey
16         lock mode is manual
17         file status is test-stat.
18 data division.
19 file section.
20 fd  counter.
21 01  counter-satz.
22     05  schluessel          pic 999.
23     05  version             pic 9(5).
24     05  wert                pic 9(5).
25 fd  testdatei.
26 01  testsatz.
27     05  testkey             pic 9(5).
```

```
28      05  testdaten           pic x(20).
29  working-storage section.
30  01  count-stat              pic xx.
31      88  count-ok            value "00" thru "09".
32  01  test-stat               pic xx.
33      88  test-ok             value "00" thru "09".
34  01  anzahlFehlversuche      pic 9(5) value 0.
35  01  anzahlVersionskonflikte pic 9(5) value 0.
36  01  ws-counter-satz.
37      05  ws-schluessel       pic 999.
38      05  ws-version          pic 9(5).
39      05  ws-wert             pic 9(5).
40  procedure division.
41  counter-auslesen.
42      open i-o
43           sharing with all other
44           retry forever
45           counter testdatei
46      if count-ok and test-ok
47          move 100 to schluessel
48          perform 10000 times
49              read counter into ws-counter-satz
50                  invalid key
51                      perform counter-satz-erzeugen
52              end-read
53              if count-ok
54                  perform testsatz-erzeugen
55              else
56                  display "Read counter:" count-stat
57              end-if
58          end-perform
59          close counter testdatei
60      else
61          display "Open counter:" count-stat
62                  " Open test: " test-stat
63      end-if
64      display "Anzahl Fehlversuche:"
65          anzahlFehlversuche
66      display "Anzahl Versionskonflikte:"
67          anzahlVersionskonflikte
68      stop run.
69  counter-satz-erzeugen.
70      *> Wenn die Datei noch nicht vorhanden war, wird
71      *> hier der Datensatz mit der Nummer 100 und dem
72      *> Anfangswert 1 erzeugt.
73      move 1 to wert version
74      write counter-satz
75      if not count-ok
```

```
 76         display "Write-counter:"
 77             count-stat
 78     end-if
 79     .
 80 testsatz-erzeugen.
 81     move ws-wert to testkey
 82     move "Testdaten" to testdaten
 83     add 1 to ws-wert
 84     *> Nachdem die Änderungen durchgeführt wurden,
 85     *> wird der Counter noch einmal gelesen und
 86     *> diesmal gesperrt.
 87     read counter with lock
 88     if count-ok
 89         *> Wenn die Versionsnummer passt, können
 90         *> alle Änderungen durchgeführt und der
 91         *> Testsatz geschrieben werden. Ansonsten
 92         *> wird der Versuch in diesem Beispiel
 93         *> einfach abgebrochen.
 94         if version = ws-version
 95             write testsatz
 96             if test-ok
 97                 display "ok:" testkey
 98             else
 99                 display "Write-testsatz:"
100                     test-stat
101                 add 1 to anzahlFehlversuche
102             end-if
103             move ws-wert to wert
104             add 1 to version
105             rewrite counter-satz
106         else
107             display "Ungültige Version"
108             add 1 to anzahlVersionskonflikte
109             unlock counter
110         end-if
111     else
112         display "Read counter:" count-stat
113     end-if
114     .
```

*Listing 3: Konkurrierender Zugriff mit optimistischen Satzsperren*

Obwohl es sich um eine reine Batch-Anwendung handelt, ist es dennoch erstaunlich, wie selten es vorkommt, dass die gelesene Version nicht mit der aktuellen übereinstimmt, wenn man das Programm zweifach parallel auf demselben Rechner startet. Bei immerhin 20.000 Zugriffen passiert es etwa nur 60- bis 70-mal.