

AppendByHistory 效能優化與索引根因

日期 2026-06-06 | DOMCMP/FORCMP 補貨比對 | 235s→16s(~14x) + 修復跨站筆數不一致 | 含核心概念

一、訴求與現象

補貨比對核心函式 AppendByHistory() 很慢：網路工作站 (S004) 跑「全部零件、105 天歷史」一次 約 235 秒。另有詭異現象：同程式同資料同天數，主機 S001=262 / 網路 S004=1977 筆數不一致。

二、先量再改 — 定位真正瓶頸

不憑空猜。先加分段計時 log跑真實資料，數字立刻說話：走分支C、setup 0.13s、loop 235s、nI=57563(掃全部零件)、nK=1977 → 96% 是「零訂單零件」的空 dbSeek，每筆 2 次網路查詢。

三、核心概念：為什麼能「減少網路查詢次數」(不是 cache)

這是整個優化的靈魂。先澄清：這不是 cache。Cache 是「把昂貴查詢的結果存起來重複用」；網路本來就不能靠 cache 救。我用的是三個概念，全指向「一開始就不要問那麼多次」：

概念	原本(慢)	改後(快)	為什麼省
反轉迴圈方向 (別大海撈針)	拿著 5.7 萬個「零件」逐一問網路: 這零件有沒有單?	直接讀「訂單」(才幾千張), 攤開就知哪些零件有單	針很少就直接數針, 別翻遍整片海。0(全部零件)→0(有訂單零件)
順序讀 vs 隨機查 (網路來回次數)	dbSeek 隨機查=每次一趟網路來回, 11.5 萬次=11.5萬趟	順序掃描=網路一整塊搬(read-ahead, 一趟50筆)	同樣資料量, 順序讀的「網路來回次數」遠少於隨機查
一次彙總進 RAM (算一次不回頭)	每個零件再回網路重讀它的訂單算數量(nQty)	順序掃描一次就把 nQty 累加進記憶體 hash, 主迴圈從 RAM 拿	不是存結果重用, 而是同一趟把該算的算完; RAM 拿值 vs 網路差好幾個數量級

倉庫比喻：原本＝拿著 5.7 萬張零件卡，每張都跑一趟倉庫問「這 105 天有沒有進出貨？」(96% 答沒有)。改法＝直接去倉庫把這 105 天的單據整疊抱回來(整疊搬、不是一張張拿)，攤開看就知道哪些零件有單，順手把數量加好放口袋。

一句話：我沒有讓網路變快、也沒有 cache 網路。我是「重新安排工作順序」，讓需要問網路的次數，從『每個零件問一次（隨機、5.7萬次）』變成『把訂單順序搬一次（幾千筆、幾十趟）』。

「放記憶體」具體指什麼？＝一個 Harbour HASH (hb_Hash)。它是住在 Stk32 程式 RAM 裡的 key-value 結構：

結構	key	value	由誰填 / 給誰用
hOrdParts (hash)	零件編號 cPartsId	該零件 105 天的 nQty(進銷加總、退貨扣掉, 已 算好)	build-S 順序掃訂單時邊掃邊 hOrdParts[cPid] += nSgn*nQty 累加; 主迴圈用 hOrdParts[cPartsId] 直接拿
aOrdKeys (array)	—	排序好的「要 loop 的有訂單零件」清單	hb_HKeys(hOrdParts)+A Sort 取得; 主迴圈逐一 dbSeek 定位 Parts

主迴圈要 nQty 時是去 RAM 查

hash (O(1)、微秒級)，不是回網路重讀（毫秒級＋來回）——省的本質就是 RAM 存取 vs 網路存取，差好幾個數量級。

與 cache 的差別：這 hash 每次跑都「現建、用完就丟」，不跨次保存。它不是「記住網路答案下次重用」，而是「這一趟把該算的算進 RAM，主迴圈就不必再回頭問網路」。

四、解決方法 — 三段遞進（概念落地）

階段	對應概念	做什麼	235s →
預過濾	概念	先建「有訂單零件」集合, 跳過 96% 空掃	97s
只 loop 有訂單零件	概念 +	不走訪全 5.7萬 Parts, 改 hb_HKeys+ASort+dbSeek 只訪有訂單的	63s
build-S 算 nQty	概念 +	順序掃一次就用 hash 累加 nQty, 主迴圈不再逐零件重掃	43s

五、順帶修復：跨站筆數不一致（索引格式坑）

根因＝訂單索引 key 用 DTOC(dActDate)(格式相依!)。索引共用、clip_date 卻各站不同(S001=yyyy/mm/dd、S004=yyyy. mm. dd)。S004

用「.」去查「/」建的索引→seek

落點偏移到年初→多撈舊單→1977。鐵證：同一份程式, S004 重建索引前 1977、後 262。

Option B 在 build-S 累加時用「日期值比較」(dActDate >= dBeginDate)跟格式無關→兩站永遠一致 262。

六、實測結果

函式 / 站台	原版	Option B	加速	筆數(改前/後)
DOMCMP / S004(網路)	235s	約 16s	~14x	262 / 262 一致
DOMCMP / S001(本機)	—	0.79s	—	262 / 262 一致
FORCMP / S004(網路)	184s	16.0s	~11x	262 / 262 一致
FORCMP / S001(本機)	3.7s	0.79s	~5x	262 / 262 一致

nK/line 改前後完全一致=輸出正確；nJ 從 343 降到 0=Option B 簽名(不再逐零件重掃)。

七、使用技術與系統性待辦

類別	內容
方法論	先量再改:分段計時 log 定位;每刀 nK/line(正確)+loop(有效) 驗證;ord/skip/build/main 拆解找下個瓶頸。
改碼	Python bytes-only 改 BIG5(避免亂碼);segment-slice 只在函式區段 replace+assert count==1;每版 .bak+VER40s/VER60s 可回溯。
Harbour	hb_Hash/hb_HKeys/hb_HHasKey、ASort(保持輸出順序)、dbSeek/dbSkip、DTC(格式相依坑) vs DTOS(固定8碼免疫)。
★ 系統性待辦	DTC 日期索引坑不只這兩支。盤點全系統 INDEX ON..DTC(..date..) → 改 DTOS + 重建索引,統一 clip_date(標準 yyyy.mm.dd),徹底拔除跨站筆數不一致。